BOOK 13: NODE.JS FUNDAMENTALS
Chapter 1: What is Node.js

JAVASCRIPT BEYOND THE BROWSER

JavaScript was created for web browsers.

For years, it could only run inside browsers like Chrome, Firefox, or Safari.

Node.js changed that. It lets JavaScript run anywhere.

With Node.js, JavaScript runs on your computer directly, on servers, on embedded devices, anywhere.

WHAT NODE.JS ACTUALLY IS

Node.js is a JavaScript runtime.

A runtime is an environment that executes code.

Browsers have a JavaScript runtime built in. Node.js provides a runtime outside browsers.

Node.js is built on V8, the same JavaScript engine that powers Chrome. Google built V8 to be incredibly fast.

Node.js takes V8 and adds things browsers do not have:
- File system access
- Network capabilities
- Operating system access
- Process management

WHY NODE.JS MATTERS

Before Node.js, you needed different languages:
- JavaScript for frontend
- PHP, Python, Ruby, Java for backend

With Node.js, you use JavaScript everywhere.

One language for:
- Web browsers
- Web servers
- Command-line tools
- Desktop applications
- Mobile applications
- IoT devices

Your JavaScript skills now apply to the entire stack.

NODE.JS IS NOT A LANGUAGE

Node.js is not a new programming language.

It is JavaScript. The same JavaScript you already know.

The difference is what Node.js provides:
- Browser JavaScript has window, document, DOM
- Node.js has process, filesystem, network

Same language, different environment, different capabilities.

EVENT-DRIVEN ARCHITECTURE

Node.js uses an event-driven model.

Instead of waiting for operations to complete, Node.js continues executing other code. When an operation finishes, it triggers an event.

Traditional approach:
1. Read file (wait)
2. Process file
3. Read another file (wait)
4. Process another file

Node.js approach:
1. Start reading file
2. Start reading another file
3. When first file ready, process it
4. When second file ready, process it

Both files are read simultaneously. Much faster.

NON-BLOCKING I/O

I/O means Input/Output. Reading files, network requests, database queries.

Blocking I/O waits for each operation:

```
let data1 = readFileSync("file1.txt")
let data2 = readFileSync("file2.txt")
```

Non-blocking I/O continues immediately:

```
readFile("file1.txt", (data) => process(data))
readFile("file2.txt", (data) => process(data))
```

Node.js defaults to non-blocking. This makes it efficient for I/O-heavy applications.

SINGLE THREADED BUT CONCURRENT

Node.js runs JavaScript on a single thread.

But it handles many operations concurrently through its event loop.

Think of a chef in a kitchen:
- One chef (single thread)
- Multiple dishes cooking simultaneously
- Chef checks each dish when ready
- Does not stand watching one pot boil

The chef is efficient by not waiting. Node.js is efficient the same way.

THE EVENT LOOP

The event loop is how Node.js achieves concurrency.

It continuously:
1. Checks if there is code to execute
2. Checks if there are callbacks ready
3. Checks if there are I/O operations complete
4. Executes what is ready
5. Repeats

As long as there is work to do, the event loop keeps running. When nothing is left, the program exits.

WHAT NODE.JS IS GOOD FOR

Web servers handling many connections:
Node.js excels at handling thousands of simultaneous connections.

Real-time applications:
Chat applications, live updates, collaborative tools.

API servers:
RESTful APIs, GraphQL servers.

Command-line tools:
Build tools, automation scripts, development utilities.

Streaming applications:
Video streaming, audio processing, file processing.

WHAT NODE.JS IS NOT IDEAL FOR

CPU-intensive tasks:
Heavy calculations block the single thread. Image processing, video encoding, complex math.

For these, use:
- Worker threads
- Child processes
- Different languages

Node.js shines with I/O. It struggles with computation.

NODE.JS ECOSYSTEM

NPM is the Node Package Manager.

It is the largest package ecosystem in the world. Over a million packages.

Need anything? There is probably a package:
- Web frameworks (Express, Fastify)
- Database drivers (MongoDB, PostgreSQL)
- Utilities (Lodash, Moment)
- Testing (Jest, Mocha)
- Build tools (Webpack, Babel)

You rarely build from scratch. You compose from packages.

HISTORY

2009 - Ryan Dahl creates Node.js
2010 - NPM launched
2011 - Windows support added
2012 - Node.js gains popularity
2014 - io.js fork happens
2015 - Node.js Foundation formed, io.js merges back
2019 - OpenJS Foundation
2023 - Node.js 20 LTS

Node.js is mature, stable, and widely used.

WHO USES NODE.JS

Netflix - streaming service
PayPal - payment processing
LinkedIn - social network
Uber - transportation
NASA - space missions
Walmart - retail giant

Used by startups and enterprises. Proven at massive scale.

NODE VS BROWSER

What Node.js has that browsers do not:
- fs (file system)
- http (create servers)
- path (file paths)
- os (operating system info)
- process (current process)
- child_process (run other programs)
- crypto (encryption)
- stream (data streaming)

What browsers have that Node.js does not:
- window
- document
- DOM
- localStorage
- fetch (added in Node 18+)
- WebSockets (need library)

Different environments, different capabilities.

PRACTICE

Understand Node.js is JavaScript outside browsers.

Understand event-driven, non-blocking architecture.

Know what Node.js is good for and not good for.

Understand the event loop concept.

WHAT YOU LEARNED

You learned Node.js runs JavaScript outside browsers.
You learned it uses V8 engine from Chrome.
You learned about event-driven architecture.
You learned about non-blocking I/O.
You learned the event loop concept.
You learned what Node.js is good for.

The next chapter teaches you how to set up Node.js.

BOOK 13: NODE.JS FUNDAMENTALS
Chapter 2: Setting Up Node.js

DOWNLOAD FROM OFFICIAL SITE

Go to nodejs.org

You will see two versions:
- LTS (Long Term Support) - Recommended for most users
- Current - Latest features, may be less stable

Download LTS for production work.

The installer works for Windows, macOS, and Linux.

INSTALLING ON WINDOWS

Download the .msi installer from nodejs.org.

Run the installer.

Click through the wizard. Default options are fine.

The installer adds Node.js to your PATH automatically.

Open Command Prompt or PowerShell:

node --version

If you see a version number, Node.js is installed.

INSTALLING ON MACOS

Option 1 - Download from nodejs.org:
Download the .pkg installer. Run it. Follow the wizard.

Option 2 - Use Homebrew:

brew install node

Homebrew keeps Node.js updated easily:

brew upgrade node

INSTALLING ON LINUX

Ubuntu/Debian:

curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs

Breaking down:

curl -fsSL - Download script. -f = fail silently. -s = silent. -S = show errors. -L = follow redirects.
setup_20.x - Setup script for Node.js version 20

| sudo -E bash - - Pipe to bash and execute with sudo. -E = preserve environment.
apt install nodejs - Install the nodejs package

Fedora:

sudo dnf install nodejs

USING NVM (NODE VERSION MANAGER)

NVM lets you install multiple Node.js versions.

Install NVM on macOS/Linux:

curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.0/install.sh | bash

Close and reopen your terminal.

Install Node.js versions:

nvm install 20
nvm install 18
nvm install 16

Switch between versions:

nvm use 20
nvm use 18

Set default version:

nvm alias default 20

List installed versions:

nvm ls

NVM is essential for working on multiple projects with different Node.js requirements.

NVM ON WINDOWS

Windows uses nvm-windows, a separate project.

Download from: github.com/coreybutler/nvm-windows

Install the .exe file.

Commands are similar:

nvm install 20
nvm use 20
nvm list

VERIFY INSTALLATION

Open terminal and run:

node --version

Output: v20.10.0 (or your version)

Also verify npm (comes with Node.js):

npm --version

Output: 10.2.3 (or your version)

Both commands should show version numbers.

THE NODE REPL

REPL stands for Read-Eval-Print-Loop.

It is an interactive JavaScript shell.

Start it:

node

You see a prompt:

>

Type JavaScript:

```
> 2 + 2
4
> "hello".toUpperCase()
'HELLO'
> const x = 10
undefined
> x * 5
50
```

Exit with:

.exit

Or press Ctrl + C twice.

REPL COMMANDS

Special commands in REPL:

.help - Show available commands
.exit - Exit the REPL
.save filename - Save current session to file
.load filename - Load and run a file
.break - Exit current expression
.clear - Reset context

Example:

```
> let arr = [1, 2, 3]
> arr.map(x => x * 2)
[2, 4, 6]
> .save session.js
Session saved to session.js
```

YOUR FIRST NODE PROGRAM

Create a file called hello.js:

```
console.log("Hello from Node.js!")
```

Run it:

```
node hello.js
```

Output: Hello from Node.js!

You just ran JavaScript outside a browser.

RUNNING JAVASCRIPT FILES

Basic run:

node filename.js

Run with arguments:

node app.js arg1 arg2

Arguments are available in process.argv:

console.log(process.argv)

Output: ['node', '/path/to/app.js', 'arg1', 'arg2']

CHECKING NODE.JS DETAILS

Get more information:

node -e "console.log(process.versions)"

Output shows versions of Node.js components:
{
 node: '20.10.0',
 v8: '11.3.244.8',
 uv: '1.46.0',
 ...
}

Check architecture:

node -p "process.arch"

Output: x64 or arm64

UPDATING NODE.JS

With installer:
Download new version from nodejs.org and install.

With Homebrew:

brew upgrade node

With NVM:

nvm install 20 --reinstall-packages-from=current

The --reinstall-packages-from flag moves your global packages to the new version.

EDITOR SETUP

Use VS Code with these extensions:
- ESLint
- Prettier
- Node.js Extension Pack

VS Code has excellent Node.js debugging built in.

Create .vscode/settings.json:

```
{
 "editor.formatOnSave": true,
 "editor.defaultFormatter": "esbenp.prettier-vscode"
}
```

PRACTICE

Install Node.js on your system.

Verify with node --version.

Try the REPL.

Create and run a JavaScript file.

Consider using NVM for version management.

WHAT YOU LEARNED

You learned to install Node.js on different systems.
You learned about LTS vs Current versions.
You learned to use NVM for version management.
You learned to verify your installation.
You learned to use the REPL.
You learned to run JavaScript files.

The next chapter teaches you how to run JavaScript with Node.js.

BOOK 13: NODE.JS FUNDAMENTALS
Chapter 3: Running JavaScript

RUNNING A FILE

Create app.js:

```
let name = "Alice"
let greeting = `Hello, ${name}!`

console.log(greeting)
```

Run it:

```
node app.js
```

Output: Hello, Alice!

Node.js reads the file, executes the JavaScript, and exits.

RUNNING INLINE CODE

Use -e flag:

node -e "console.log(2 + 2)"

Output: 4

Breaking down:

node - The Node.js command
-e "code" - Execute flag. -e = evaluate. The string is JavaScript code.

Useful for quick one-liners.

PRINTING RESULT WITH -p

Use -p flag to print the result:

node -p "Math.random()"

Output: 0.7234567890123456

Breaking down:

node - The Node.js command
-p "expression" - Print flag. Evaluates and prints the result.

Difference:
-e executes but does not print the result unless you use console.log
-p automatically prints the final value

COMMAND LINE ARGUMENTS

Access arguments via process.argv:

Create args.js:

```
console.log(process.argv)
```

Run:

```
node args.js hello world 123
```

Output:

```
[
 '/usr/local/bin/node',
 '/path/to/args.js',
 'hello',
 'world',
 '123'
]
```

Index 0 is node executable path.
Index 1 is script path.
Index 2+ are your arguments.

Get just your arguments:

```
let args = process.argv.slice(2)

console.log(args)
```

Output: ['hello', 'world', '123']

PRACTICAL ARGUMENT PARSING

Create greet.js:

```
let args = process.argv.slice(2)
let name = args[0] ?? "World"

console.log(`Hello, ${name}!`)
```

Run:

```
node greet.js
node greet.js Alice
node greet.js Bob
```

Output:
Hello, World!
Hello, Alice!
Hello, Bob!

EXIT CODES

Programs return exit codes. 0 means success. Non-zero means error.

Exit explicitly:

```
process.exit(0)
process.exit(1)
```

Check exit code in terminal:

```
node app.js
echo $?
```

Output shows the exit code.

Use exit codes for scripts:

```
if (error) {
 console.error("Something went wrong")
 process.exit(1)
}

console.log("Success")
process.exit(0)
```

STANDARD INPUT AND OUTPUT

Node.js has three standard streams:

process.stdin - Input stream
process.stdout - Output stream
process.stderr - Error stream

console.log writes to stdout.
console.error writes to stderr.

Write directly:

```
process.stdout.write("Hello")
process.stdout.write(" World\n")
```

Output: Hello World

No automatic newline with process.stdout.write.

READING FROM STDIN

Read user input:

```
process.stdin.setEncoding("utf8")

process.stdin.on("data", (data) => {
 console.log("You typed:", data.trim())
})

console.log("Type something and press Enter:")
```

Run:

```
node input.js
```

Type something and press Enter.

READLINE FOR INTERACTIVE INPUT

Use readline module:

```
const readline = require("readline")

let rl = readline.createInterface({
 input: process.stdin,
 output: process.stdout
})

rl.question("What is your name? ", (answer) => {
 console.log(`Hello, ${answer}!`)
 rl.close()
})
```

Run:

```
node question.js
What is your name? Alice
Hello, Alice!
```

MULTIPLE QUESTIONS

```
const readline = require("readline")

let rl = readline.createInterface({
 input: process.stdin,
 output: process.stdout
})

function ask(question) {
 return new Promise((resolve) => {
 rl.question(question, resolve)
 })
}

async function main() {
 let name = await ask("Name: ")
 let age = await ask("Age: ")
 let city = await ask("City: ")

 console.log(`${name}, ${age} years old, from ${city}`)

 rl.close()
}

main()
```

ENVIRONMENT VARIABLES

Access environment variables:

```
console.log(process.env.PATH)
console.log(process.env.HOME)
```

console.log(process.env.USER)

Set environment variables when running:

API_KEY=abc123 node app.js

Access in code:

let apiKey = process.env.API_KEY

console.log(apiKey)

Output: abc123

THE GLOBAL OBJECT

Browser has window. Node.js has global.

console.log(global)

You rarely use global directly. These are available globally:
- console
- setTimeout, setInterval
- process
- Buffer
- require (CommonJS)

In Node.js, variables you declare are NOT global:

let x = 10

This x is local to the file, not on global.

CURRENT DIRECTORY AND FILE

Special variables for file location:

_filename - Full path to current file
_dirname - Directory of current file

console.log(_filename)
console.log(_dirname)

Output:
/home/user/project/app.js
/home/user/project

These work in CommonJS modules. ES modules use import.meta.url instead.

RUNNING WITH DIFFERENT OPTIONS

Check syntax without running:

node --check app.js

Increase memory limit:

node --max-old-space-size=4096 app.js

Enable experimental features:

node --experimental-modules app.js

Show all options:

node --help

WATCH MODE

Automatically restart on file changes (Node.js 18+):

node --watch app.js

Every time you save app.js, Node.js restarts.

Great for development.

PRACTICE

Run JavaScript files with node.

Use command line arguments.

Read user input with readline.

Access environment variables.

Know _filename and _dirname.

WHAT YOU LEARNED

You learned to run JavaScript files.
You learned inline execution with -e and -p.
You learned command line arguments.
You learned exit codes.
You learned stdin/stdout/stderr.
You learned readline for input.
You learned environment variables.
You learned _filename and _dirname.

The next chapter teaches you about modules and require.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 4: Modules and Require

WHY MODULES

Without modules, all code goes in one file.

Problems:
- Files become huge
- Name conflicts
- Hard to organize
- Hard to reuse
- Hard to maintain

Modules split code into separate files.

Each file is a module. Each module has its own scope.

COMMONJS MODULES

Node.js originally used CommonJS module system.

Export with module.exports.
Import with require.

This is the traditional Node.js way. Still widely used.

CREATING A MODULE

Create math.js:

```
function add(a, b) {
 return a + b
}

function subtract(a, b) {
 return a - b
}

function multiply(a, b) {
 return a * b
}

module.exports = { add, subtract, multiply }
```

The module.exports object is what other files receive when they require this module.

USING A MODULE

Create app.js:

```
const math = require("./math")

console.log(math.add(2, 3))
console.log(math.subtract(10, 4))
console.log(math.multiply(5, 6))
```

Output:
5
6
30

require returns the module.exports object from the required file.

THE REQUIRE PATH

Relative paths start with ./ or ../:

```
const math = require("./math")
const utils = require("../utils/helpers")
```

Without ./ it looks for:
1. Built-in Node.js modules
2. node_modules folder

```
const fs = require("fs")
const lodash = require("lodash")
```

FILE EXTENSIONS

You can omit .js extension:

```
const math = require("./math")
```

Same as:

```
const math = require("./math.js")
```

Node.js tries these extensions:
1. .js
2. .json
3. .node

For clarity, many developers include the extension.

EXPORTING A SINGLE FUNCTION

Create greet.js:

```
module.exports = function(name) {
 return `Hello, ${name}!`
}
```

Use it:

```
const greet = require("./greet")

console.log(greet("Alice"))
```

Output: Hello, Alice!

module.exports can be anything: function, object, class, string.

EXPORTING A CLASS

Create user.js:

```
class User {
 constructor(name, email) {
 this.name = name
 this.email = email
 }

 greet() {
 return `Hello, I am ${this.name}`
 }
}

module.exports = User
```

Use it:

```
const User = require("./user")
```

```
let alice = new User("Alice", "alice@example.com")

console.log(alice.greet())
```

EXPORTS SHORTHAND

exports is a shorthand for module.exports:

```
exports.add = function(a, b) {
 return a + b
}

exports.PI = 3.14159
```

Same as:

```
module.exports.add = function(a, b) {
 return a + b
}

module.exports.PI = 3.14159
```

Warning: Do not assign to exports directly:

```
exports = { add }
```

This does NOT work. It breaks the reference.

Always use module.exports for direct assignment.

DESTRUCTURING REQUIRE

Extract specific exports:

```
const { add, subtract } = require("./math")

console.log(add(2, 3))
console.log(subtract(10, 4))
```

Cleaner than accessing through an object.

MODULE CACHING

Modules are cached after first require.

Create counter.js:

```
let count = 0

function increment() {
 count++
 return count
}

module.exports = { increment }
```

Create app.js:

```
const counter1 = require("./counter")
const counter2 = require("./counter")

console.log(counter1.increment())
console.log(counter1.increment())
console.log(counter2.increment())
```

Output:
1
2
3

Both counter1 and counter2 reference the same cached module.

The module code runs once. Future requires return the cached exports.

CLEARING CACHE

Force reload a module:

```
delete require.cache[require.resolve("./counter")]
const counter = require("./counter")
```

Rarely needed. Usually caching is what you want.

CIRCULAR DEPENDENCIES

Modules can require each other:

a.js:

```
const b = require("./b")

console.log("a loaded")

module.exports = { name: "a" }
```

b.js:

```
const a = require("./a")

console.log("b loaded")

module.exports = { name: "b" }
```

This works but can cause issues. a gets incomplete exports from b.

Avoid circular dependencies when possible.

BUILT-IN MODULES

Node.js includes many modules:

```
const fs = require("fs")
const path = require("path")
const http = require("http")
const os = require("os")
const crypto = require("crypto")
const events = require("events")
```

No installation needed. Just require them.

CHECKING IF MODULE IS MAIN

Check if current file is the entry point:

```
if (require.main === module) {
 console.log("This file is run directly")
} else {
 console.log("This file is required by another")
}
```

Useful for modules that can also run standalone:

```
function greet(name) {
 return `Hello, ${name}!`
}

module.exports = greet
```

```
if (require.main === module) {
 console.log(greet("World"))
}
```

MODULE WRAPPER

Node.js wraps every module in a function:

```
(function(exports, require, module, _filename, _dirname) {
 // Your module code here
})
```

This is why each module has its own scope.

This is why require, _filename, _dirname are available.

They are function parameters, not globals.

FOLDER AS MODULE

Create a folder utils with index.js inside:

```
utils/
 index.js
 helpers.js
 validators.js
```

utils/index.js:

```
const helpers = require("./helpers")
const validators = require("./validators")

module.exports = { ...helpers, ...validators }
```

Now require the folder:

```
const utils = require("./utils")
```

Node.js automatically loads index.js from folders.

PRACTICE

Create modules with exports.

Require modules in other files.

Understand module caching.

Use built-in modules.

Organize code with folder modules.

WHAT YOU LEARNED

You learned CommonJS module system.
You learned module.exports and require.
You learned relative and absolute paths.
You learned exporting functions, objects, classes.
You learned module caching.
You learned to check if module is main.
You learned about the module wrapper.

The next chapter teaches you about ES Modules.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 5: ES Modules

ES MODULES VS COMMONJS

CommonJS:
- Original Node.js module system
- Uses require and module.exports
- Synchronous loading
- Dynamic imports possible

ES Modules (ESM):
- JavaScript standard module system
- Uses import and export
- Asynchronous loading
- Static analysis possible

ES Modules are the future. CommonJS still works.

ENABLING ES MODULES

Option 1 - Use .mjs extension:

math.mjs
app.mjs

Option 2 - Set type in package.json:

```
{
 "type": "module"
}
```

All .js files in that project become ES modules.

Option 3 - Use .cjs for CommonJS in ESM project:

utils.cjs

NAMED EXPORTS

Create math.mjs:

```
export function add(a, b) {
 return a + b
}

export function subtract(a, b) {
 return a - b
}

export const PI = 3.14159
```

Or export at the end:

```
function add(a, b) {
 return a + b
}

function subtract(a, b) {
 return a - b
}

const PI = 3.14159

export { add, subtract, PI }
```

IMPORTING NAMED EXPORTS

```
import { add, subtract, PI } from "./math.mjs"

console.log(add(2, 3))
console.log(subtract(10, 4))
```

console.log(PI)

Curly braces for named imports.

Names must match what was exported.

DEFAULT EXPORTS

Create greet.mjs:

```
export default function(name) {
 return `Hello, ${name}!`
}
```

Or:

```
function greet(name) {
 return `Hello, ${name}!`
}

export default greet
```

One default export per module.

IMPORTING DEFAULT EXPORTS

```
import greet from "./greet.mjs"

console.log(greet("Alice"))
```

No curly braces for default imports.

You can name it anything:

```
import sayHello from "./greet.mjs"

console.log(sayHello("Bob"))
```

MIXED EXPORTS

Create user.mjs:

```
export default class User {
 constructor(name) {
 this.name = name
 }
}

export function createUser(name) {
 return new User(name)
}

export const DEFAULT_ROLE = "user"
```

Import:

```
import User, { createUser, DEFAULT_ROLE } from "./user.mjs"
```

RENAMING IMPORTS

```
import { add as sum, subtract as minus } from "./math.mjs"

console.log(sum(2, 3))
console.log(minus(10, 4))
```

IMPORT ALL

```
import * as math from "./math.mjs"

console.log(math.add(2, 3))
console.log(math.subtract(10, 4))
console.log(math.PI)
```

Creates a namespace object with all exports.

RE-EXPORTING

Export from another module:

```
export { add, subtract } from "./math.mjs"

export { default } from "./greet.mjs"

export { default as greet } from "./greet.mjs"

export * from "./utils.mjs"
```

Useful for index files:

index.mjs:

```
export { User } from "./user.mjs"
export { Product } from "./product.mjs"
export { Order } from "./order.mjs"
```

Import from one place:

```
import { User, Product, Order } from "./models/index.mjs"
```

DYNAMIC IMPORTS

Import modules at runtime:

```
async function loadModule() {
 let module = await import("./math.mjs")

 console.log(module.add(2, 3))
}

loadModule()
```

Dynamic imports return a promise.

Works in both CommonJS and ESM.

CONDITIONAL IMPORTS

```
let modulePath = process.env.NODE_ENV === "production"
 ? "./prod-config.mjs"
 : "./dev-config.mjs"

let config = await import(modulePath)
```

TOP-LEVEL AWAIT

ES modules support await at the top level:

```
let response = await fetch("https://api.example.com/data")
let data = await response.json()

export { data }
```

No need to wrap in async function.

Only works in ES modules.

NO _FILENAME AND _DIRNAME

ES modules do not have _filename and _dirname.

Use import.meta.url instead:

console.log(import.meta.url)

Output: file:///home/user/project/app.mjs

Get directory and filename:

```
import { fileURLToPath } from "url"
import { dirname } from "path"

const _filename = fileURLToPath(import.meta.url)
const _dirname = dirname(_filename)

console.log(_filename)
console.log(_dirname)
```

IMPORTING JSON

In ES modules, JSON requires assertion:

```
import config from "./config.json" assert { type: "json" }
```

Or use createRequire:

```
import { createRequire } from "module"
```

```
const require = createRequire(import.meta.url)
const config = require("./config.json")
```

IMPORTING COMMONJS FROM ESM

ES modules can import CommonJS:

```
import lodash from "lodash"
```

The default export is module.exports.

Named exports may not work:

```
import { map } from "lodash"
```

May need:

```
import lodash from "lodash"
const { map } = lodash
```

IMPORTING ESM FROM COMMONJS

CommonJS cannot use import syntax.

Use dynamic import:

```
async function main() {
 const module = await import("./esm-module.mjs")

 console.log(module.default)
}

main()
```

PACKAGE.JSON EXPORTS

Control what can be imported:

```
{
 "name": "my-package",
 "exports": {
 ".": "./src/index.mjs",
 "./utils": "./src/utils.mjs"
 }
}
```

Users can import:

```
import main from "my-package"
import utils from "my-package/utils"
```

They cannot import internal files.

WHICH TO USE

For new projects:
Use ES Modules with "type": "module" in package.json.

For existing projects:
Stick with CommonJS unless you have a reason to switch.

For libraries:
Consider supporting both with build tools.

The ecosystem is moving to ESM. CommonJS will work for a long time.

PRACTICE

Create ES modules with named and default exports.

Import using various syntaxes.

Use dynamic imports.

Handle _dirname and _filename in ESM.

Understand the differences from CommonJS.

WHAT YOU LEARNED

You learned ES Modules syntax.
You learned named and default exports.
You learned various import syntaxes.
You learned dynamic imports.
You learned top-level await.
You learned import.meta.url for file paths.
You learned interoperability with CommonJS.

The next chapter teaches you about NPM package manager.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 6: NPM Package Manager

WHAT IS NPM

NPM is Node Package Manager.

It does three things:

1. Downloads packages from npmjs.com
2. Manages project dependencies
3. Runs project scripts

NPM comes installed with Node.js.

INITIALIZING A PROJECT

Create package.json:

npm init

Answer the prompts or accept defaults.

Skip prompts with:

npm init -y

Breaking down:

npm init - Initialize a new Node.js project
-y - Yes flag. Accept all defaults.

Creates package.json with project information.

PACKAGE.JSON STRUCTURE

```
{
 "name": "my-project",
 "version": "1.0.0",
 "description": "My Node.js project",
 "main": "index.js",
 "scripts": {
 "start": "node index.js",
```

```
 "test": "jest"
 },
 "dependencies": {
 "express": "^4.18.2"
 },
 "devDependencies": {
 "jest": "^29.7.0"
 }
}
```

name - Package name
version - Current version
main - Entry point file
scripts - Commands to run
dependencies - Required packages
devDependencies - Development-only packages

INSTALLING PACKAGES

Install a package:

```
npm install lodash
```

Breaking down:

npm install - Install command
lodash - Package name

This:
1. Downloads lodash from npmjs.com
2. Adds it to dependencies in package.json
3. Creates node_modules folder
4. Creates package-lock.json

Short form:

```
npm i lodash
```

DEV DEPENDENCIES

Install for development only:

npm install --save-dev jest

Breaking down:

npm install - Install command
--save-dev - Add to devDependencies. Short form: -D
jest - Package name

Dev dependencies are not installed in production.

Short form:

npm i -D jest

INSTALLING ALL DEPENDENCIES

When you clone a project:

npm install

Reads package.json and installs everything in dependencies and devDependencies.

Production only (skip devDependencies):

npm install --production

PACKAGE-LOCK.JSON

Records exact versions of everything installed.

Ensures everyone gets the same versions.

Commit this file to version control.

Never edit it manually.

SEMANTIC VERSIONING

Versions follow Major.Minor.Patch:

1.2.3
^ Major.Minor.Patch

Major - Breaking changes
Minor - New features, backwards compatible
Patch - Bug fixes

In package.json:

^4.18.2 - Allow minor and patch updates (4.18.2 to 4.99.99)
~4.18.2 - Allow patch updates only (4.18.2 to 4.18.99)
4.18.2 - Exact version only

UPDATING PACKAGES

See outdated packages:

npm outdated

Update to allowed versions:

npm update

Update specific package:

npm update lodash

Update to latest (may have breaking changes):

npm install lodash@latest

UNINSTALLING PACKAGES

Remove a package:

npm uninstall lodash

Removes from node_modules and package.json.

Short form:

npm rm lodash

GLOBAL PACKAGES

Install globally (available everywhere):

npm install -g nodemon

Breaking down:

npm install - Install command
-g - Global flag. Install system-wide.

nodemon - Package name

Global packages provide command-line tools.

List global packages:

npm list -g --depth = 0

NPM SCRIPTS

Define commands in package.json:

```
{
 "scripts": {
 "start": "node index.js",
 "dev": "nodemon index.js",
 "test": "jest",
 "build": "webpack",
 "lint": "eslint ."
 }
}
```

Run scripts:

npm run start
npm run dev
npm run test
npm run build

start and test have shortcuts:

npm start
npm test

SCRIPT SHORTCUTS

npm start - Runs start script
npm test - Runs test script
npm stop - Runs stop script
npm restart - Runs restart script

All others need npm run:

npm run dev
npm run build
npm run lint

NPX

Run packages without installing:

npx create-react-app my-app

Breaking down:

npx - Execute package command
create-react-app - Package to run
my-app - Arguments to pass

npx downloads, runs, and cleans up.

Useful for one-time commands.

VIEWING PACKAGE INFO

See package information:

npm info lodash

See all versions:

npm info lodash versions

See package homepage:

npm docs lodash

SEARCHING PACKAGES

Search npmjs.com:

npm search http server

Or visit npmjs.com directly.

PUBLISHING PACKAGES

Create an npm account at npmjs.com.

Login:

npm login

Publish:

npm publish

Your package is now on npmjs.com.

Update version before republishing:

npm version patch

npm version minor
npm version major

NPM AUDIT

Check for security vulnerabilities:

npm audit

Fix vulnerabilities:

npm audit fix

Force fix (may have breaking changes):

npm audit fix --force

CLEANING UP

Remove unused packages:

npm prune

Clear npm cache:

npm cache clean --force

Remove node_modules and reinstall:

rm -rf node_modules
npm install

.NPMRC CONFIGURATION

Create .npmrc for npm settings:

save-exact = true
package-lock = true

Project-level or user-level (~/.npmrc).

PRACTICE

Initialize projects with npm init.

Install and manage dependencies.

Write and run npm scripts.

Understand semantic versioning.

Use npx for one-time commands.

WHAT YOU LEARNED

You learned to initialize Node.js projects.
You learned to install packages.
You learned dependencies vs devDependencies.
You learned semantic versioning.
You learned npm scripts.
You learned npx for execution.
You learned npm audit for security.

The next chapter teaches you about the process object.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 7: The Process Object

WHAT IS PROCESS

process is a global object in Node.js.

It provides information about and control over the current Node.js process.

You do not need to require it. It is always available.

```
console.log(process)
```

COMMAND LINE ARGUMENTS

process.argv is an array of command line arguments:

```
console.log(process.argv)
```

Run:

```
node app.js hello world
```

Output:

```
[
 '/usr/local/bin/node',
 '/path/to/app.js',
 'hello',
 'world'
]
```

Get user arguments:

```
let args = process.argv.slice(2)

console.log(args)
```

Output: ['hello', 'world']

PARSING ARGUMENTS

Simple named arguments:

```
function parseArgs(args) {
 let result = {}

 for (let i = 0; i < args.length; i++) {
 if (args[i].startsWith("--")) {
 let key = args[i].slice(2)
 let value = args[i + 1]

 if (value && !value.startsWith("--")) {
 result[key] = value
 i++
 } else {
 result[key] = true
 }
 }
 }

 return result
}

let options = parseArgs(process.argv.slice(2))

console.log(options)
```

Run:

```
node app.js --name Alice --verbose --port 3000
```

Output: { name: 'Alice', verbose: true, port: '3000' }

ENVIRONMENT VARIABLES

process.env contains environment variables:

```
console.log(process.env.PATH)
console.log(process.env.HOME)
console.log(process.env.NODE_ENV)
```

Set variables when running:

```
NODE_ENV=production API_KEY=secret node app.js
```

Access:

```
let env = process.env.NODE_ENV ?? "development"
let apiKey = process.env.API_KEY

console.log(`Environment: ${env}`)
```

Do not hardcode secrets. Use environment variables.

CURRENT WORKING DIRECTORY

process.cwd() returns current working directory:

```
console.log(process.cwd())
```

Output: /home/user/project

This is where you ran node from, not where the script is.

Change directory:

process.chdir("/tmp")

console.log(process.cwd())

Output: /tmp

PROCESS ID

process.pid is the process ID:

console.log(`PID: ${process.pid}`)

Useful for logging, debugging, process management.

Parent process ID:

console.log(`Parent PID: ${process.ppid}`)

EXITING

Exit the process:

process.exit()
process.exit(0)
process.exit(1)

0 means success. Non-zero means error.

Set exit code without exiting:

process.exitCode = 1

The process uses this code when it exits naturally.

EXIT EVENT

Run code before exit:

```
process.on("exit", (code) = > {
 console.log(`Exiting with code: ${code}`)
})
```

Cannot do async work in exit handler. The process is ending.

For cleanup, use beforeExit:

```
process.on("beforeExit", async (code) = > {
 await cleanup()
 console.log("Cleanup done")
})
```

UNCAUGHT EXCEPTIONS

Handle uncaught exceptions:

```
process.on("uncaughtException", (error) = > {
 console.error("Uncaught exception:", error)
 process.exit(1)
})

throw new Error("Oops!")
```

Warning: After uncaughtException, the process may be in an undefined state. Best practice is to log and exit.

UNHANDLED REJECTIONS

Handle unhandled promise rejections:

```
process.on("unhandledRejection", (reason, promise) = > {
 console.error("Unhandled rejection:", reason)
})

Promise.reject(new Error("Promise failed"))
```

SIGNALS

Respond to OS signals:

```
process.on("SIGINT", () = > {
 console.log("Received SIGINT (Ctrl + C)")
 process.exit(0)
})

process.on("SIGTERM", () = > {
 console.log("Received SIGTERM")
 gracefulShutdown()
})
```

SIGINT - Interrupt (Ctrl + C)
SIGTERM - Termination request
SIGHUP - Terminal closed

MEMORY USAGE

process.memoryUsage() returns memory information:

console.log(process.memoryUsage())

Output:
{
 rss: 30000000,
 heapTotal: 6000000,
 heapUsed: 4000000,
 external: 1000000,
 arrayBuffers: 10000
}

rss - Resident Set Size, total memory allocated
heapTotal - Total heap size
heapUsed - Heap actually used
external - Memory used by C++ objects
arrayBuffers - Memory for ArrayBuffers

CPU USAGE

process.cpuUsage() returns CPU time:

console.log(process.cpuUsage())

Output: { user: 100000, system: 50000 }

Values are in microseconds.

UPTIME

process.uptime() returns seconds since process started:

console.log(`Uptime: ${process.uptime()} seconds`)

setInterval(() => {

```
 console.log(`Running for ${process.uptime().toFixed(0)}s`)
}, 1000)
```

PLATFORM AND ARCHITECTURE

process.platform - Operating system:

console.log(process.platform)

Output: linux, darwin, win32

process.arch - CPU architecture:

console.log(process.arch)

Output: x64, arm64, arm

NODE VERSION

process.version - Node.js version:

console.log(process.version)

Output: v20.10.0

process.versions - All component versions:

console.log(process.versions)

Output:

```
{
 node: '20.10.0',
 v8: '11.3.244.8',
 uv: '1.46.0',
```

```
 ...
}
```

STANDARD STREAMS

process.stdin - Standard input
process.stdout - Standard output
process.stderr - Standard error

Write to stdout:

```
process.stdout.write("Hello ")
process.stdout.write("World\n")
```

Write to stderr:

```
process.stderr.write("Error: Something went wrong\n")
```

Read from stdin:

```
process.stdin.on("data", (data) => {
 console.log(`Input: ${data}`)
})
```

NEXT TICK

process.nextTick schedules a callback:

```
console.log("Start")

process.nextTick(() => {
 console.log("Next tick")
})
```

```
console.log("End")

Output:
Start
End
Next tick
```

nextTick runs before I/O callbacks, after current operation.

PRACTICE

Access command line arguments.

Use environment variables.

Handle exit and signals.

Monitor memory and uptime.

Use standard streams.

WHAT YOU LEARNED

You learned process.argv for arguments.
You learned process.env for environment.
You learned process.exit and exit codes.
You learned signal handling.
You learned memory and CPU monitoring.
You learned platform detection.
You learned standard streams.

The next chapter teaches you about file system basics.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 8: File System Basics

THE FS MODULE

Node.js provides the fs module for file system operations.

```
const fs = require("fs")
```

Or with promises:

```
const fs = require("fs").promises
```

Or ES modules:

```
import fs from "fs"
import { readFile, writeFile } from "fs/promises"
```

SYNC VS ASYNC

Every operation has two versions:

Synchronous - Blocks until complete
Asynchronous - Returns immediately, calls callback when done

```
fs.readFileSync("file.txt")
```

```
fs.readFile("file.txt", callback)
```

Use async for most cases. Sync blocks the entire process.

CALLBACK PATTERN

Traditional Node.js callback pattern:

```
fs.readFile("file.txt", "utf8", (error, data) = > {
 if (error) {
 console.error("Error:", error.message)
 return
 }

 console.log(data)
})
```

First argument is always error (null if success).
Second argument is the result.

Always check for errors first.

PROMISE PATTERN

Modern async/await with fs.promises:

```
const fs = require("fs").promises

async function main() {
 try {
 let data = await fs.readFile("file.txt", "utf8")
 console.log(data)
 } catch (error) {
 console.error("Error:", error.message)
 }
}

main()
```

Cleaner than callbacks. Recommended approach.

CHECKING IF FILE EXISTS

fs.existsSync checks synchronously:

```
if (fs.existsSync("file.txt")) {
 console.log("File exists")
}
```

For async, use fs.access:

```
const fs = require("fs").promises

async function fileExists(path) {
 try {
 await fs.access(path)
 return true
 } catch {
 return false
 }
}

let exists = await fileExists("file.txt")
```

FILE STATS

Get file information:

```
const fs = require("fs").promises

async function main() {
 let stats = await fs.stat("file.txt")

 console.log("Size:", stats.size, "bytes")
 console.log("Created:", stats.birthtime)
 console.log("Modified:", stats.mtime)
```

```
 console.log("Is file:", stats.isFile())
 console.log("Is directory:", stats.isDirectory())
}
```

Stats properties:

size - File size in bytes
birthtime - Creation time
mtime - Last modified time
atime - Last access time
mode - File permissions

Stats methods:

isFile() - Is it a file?
isDirectory() - Is it a directory?
isSymbolicLink() - Is it a symlink?

FILE PERMISSIONS

Check permissions:

```
const fs = require("fs").promises
const { constants } = require("fs")

async function checkPermissions(path) {
 try {
 await fs.access(path, constants.R_OK)
 console.log("Readable")
 } catch {
 console.log("Not readable")
 }

 try {
 await fs.access(path, constants.W_OK)
 console.log("Writable")
 } catch {
```

```
 console.log("Not writable")
 }

 try {
 await fs.access(path, constants.X_OK)
 console.log("Executable")
 } catch {
 console.log("Not executable")
 }
}
```

Permission constants:

constants.F_OK - File exists
constants.R_OK - File readable
constants.W_OK - File writable
constants.X_OK - File executable

RENAMING FILES

```
const fs = require("fs").promises

await fs.rename("old.txt", "new.txt")
```

Move file to different directory:

```
await fs.rename("file.txt", "backup/file.txt")
```

DELETING FILES

```
const fs = require("fs").promises

await fs.unlink("file.txt")
```

unlink deletes a file.

Check before deleting:

```
async function deleteIfExists(path) {
 try {
 await fs.unlink(path)
 console.log("Deleted:", path)
 } catch (error) {
 if (error.code !== "ENOENT") {
 throw error
 }
 }
}
```

ENOENT means file does not exist.

COPYING FILES

```
const fs = require("fs").promises

await fs.copyFile("source.txt", "destination.txt")
```

With flags:

```
const { constants } = require("fs")

await fs.copyFile("source.txt", "dest.txt", constants.COPYFILE_EXCL)
```

COPYFILE_EXCL - Fail if destination exists

COMMON ERROR CODES

ENOENT - File or directory does not exist

EEXIST - File or directory already exists
EACCES - Permission denied
EISDIR - Expected file but got directory
ENOTDIR - Expected directory but got file
ENOTEMPTY - Directory not empty
EMFILE - Too many open files

Handle errors by code:

```
try {
 await fs.readFile("file.txt")
} catch (error) {
 switch (error.code) {
 case "ENOENT":
 console.log("File not found")
 break
 case "EACCES":
 console.log("Permission denied")
 break
 default:
 throw error
 }
}
```

SYMBOLIC LINKS

Create symlink:

```
await fs.symlink("target.txt", "link.txt")
```

Read symlink target:

```
let target = await fs.readlink("link.txt")
```

Get stats following symlink:

```
let stats = await fs.stat("link.txt")
```

Get stats of symlink itself:

```
let stats = await fs.lstat("link.txt")
```

WATCHING FILES

Watch for changes:

```
fs.watch("file.txt", (eventType, filename) = > {
 console.log(`Event: ${eventType}`)
 console.log(`File: ${filename}`)
})
```

Event types:

rename - File renamed or deleted
change - File contents changed

Watch a directory:

```
fs.watch(".", (eventType, filename) = > {
 console.log(`${filename}: ${eventType}`)
})
```

PRACTICE

Use fs.promises for modern code.

Handle errors properly.

Check file existence and permissions.

Get file statistics.

Watch for file changes.

WHAT YOU LEARNED

You learned to import fs module.
You learned sync vs async operations.
You learned callback and promise patterns.
You learned to check file existence.
You learned to get file stats.
You learned to rename, delete, and copy files.
You learned common error codes.

The next chapter teaches you about reading files.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 9: Reading Files

READING TEXT FILES

Read entire file as string:

```
const fs = require("fs").promises

async function main() {
 let content = await fs.readFile("file.txt", "utf8")
 console.log(content)
}

main()
```

The second argument is encoding. utf8 for text files.

Without encoding, you get a Buffer (raw bytes).

READING WITH CALLBACKS

```
const fs = require("fs")

fs.readFile("file.txt", "utf8", (error, data) => {
 if (error) {
 console.error(error.message)
 return
 }

 console.log(data)
})
```

READING SYNCHRONOUSLY

```
const fs = require("fs")

let content = fs.readFileSync("file.txt", "utf8")

console.log(content)
```

Only use sync for startup scripts or CLI tools.

READING AS BUFFER

Without encoding, you get a Buffer:

```
let buffer = await fs.readFile("file.txt")

console.log(buffer)
```

Output: <Buffer 48 65 6c 6c 6f ...>

Convert to string:

let text = buffer.toString("utf8")

Useful for binary files or when you need raw bytes.

READING JSON FILES

Read and parse JSON:

```
async function readJson(path) {
 let content = await fs.readFile(path, "utf8")
 return JSON.parse(content)
}

let config = await readJson("config.json")

console.log(config.port)
```

Handle parse errors:

```
async function readJsonSafe(path) {
 try {
 let content = await fs.readFile(path, "utf8")
 return JSON.parse(content)
 } catch (error) {
 if (error.code === "ENOENT") {
 return null
 }
 throw error
 }
}
```

READING LINE BY LINE

Using readline module:

```
const fs = require("fs")
const readline = require("readline")

async function readLines(path) {
 let stream = fs.createReadStream(path)

 let rl = readline.createInterface({
 input: stream,
 crlfDelay: Infinity
 })

 let lines = []

 for await (let line of rl) {
 lines.push(line)
 }

 return lines
}

let lines = await readLines("file.txt")

lines.forEach((line, i) => {
 console.log(`${i + 1}: ${line}`)
})
```

PROCESSING LINES AS THEY COME

```
const fs = require("fs")
const readline = require("readline")
```

```
async function processLines(path, handler) {
 let stream = fs.createReadStream(path)

 let rl = readline.createInterface({
 input: stream,
 crlfDelay: Infinity
 })

 let lineNumber = 0

 for await (let line of rl) {
 lineNumber++
 await handler(line, lineNumber)
 }
}

await processLines("large-file.txt", (line, num) => {
 if (line.includes("error")) {
 console.log(`Line ${num}: ${line}`)
 }
})
```

READING PARTIAL FILE

Read only part of a file:

```
const fs = require("fs").promises

async function readBytes(path, start, length) {
 let handle = await fs.open(path, "r")
 let buffer = Buffer.alloc(length)

 await handle.read(buffer, 0, length, start)
 await handle.close()

 return buffer
}
```

```
let first100 = await readBytes("file.txt", 0, 100)

console.log(first100.toString())
```

READING LARGE FILES

Do not read entire large files into memory.

Bad:

```
let huge = await fs.readFile("10gb-file.txt")
```

Good - use streams:

```
const fs = require("fs")

let stream = fs.createReadStream("large-file.txt", {
 encoding: "utf8",
 highWaterMark: 64 * 1024
})

stream.on("data", (chunk) => {
 console.log("Received chunk:", chunk.length, "bytes")
})

stream.on("end", () => {
 console.log("Done reading")
})

stream.on("error", (error) => {
 console.error("Error:", error.message)
})
```

READING MULTIPLE FILES

Read in parallel:

```
async function readMultiple(paths) {
 let promises = paths.map(path => fs.readFile(path, "utf8"))
 return Promise.all(promises)
}

let [file1, file2, file3] = await readMultiple([
 "one.txt",
 "two.txt",
 "three.txt"
])
```

Read sequentially:

```
async function readSequential(paths) {
 let results = []

 for (let path of paths) {
 let content = await fs.readFile(path, "utf8")
 results.push(content)
 }

 return results
}
```

READING BINARY FILES

```
let imageBuffer = await fs.readFile("image.png")

console.log("Size:", imageBuffer.length, "bytes")

let firstBytes = imageBuffer.slice(0, 8)

console.log("First 8 bytes:", firstBytes)
```

PNG files start with specific bytes:

```
let isPng = imageBuffer[0] === 0x89 &&
 imageBuffer[1] === 0x50 &&
 imageBuffer[2] === 0x4E &&
 imageBuffer[3] === 0x47
```

READING WITH FILE HANDLES

More control with file handles:

```
const fs = require("fs").promises

async function readWithHandle(path) {
 let handle

 try {
 handle = await fs.open(path, "r")
 let stats = await handle.stat()

 console.log("Size:", stats.size)

 let content = await handle.readFile("utf8")

 return content
 } finally {
 if (handle) {
 await handle.close()
 }
 }
}
```

CHECKING BEFORE READING

```
async function safeRead(path) {
 try {
 let stats = await fs.stat(path)

 if (!stats.isFile()) {
 throw new Error("Not a file")
 }

 if (stats.size > 100 * 1024 * 1024) {
 throw new Error("File too large")
 }

 return fs.readFile(path, "utf8")
 } catch (error) {
 if (error.code === "ENOENT") {
 return null
 }
 throw error
 }
}
```

READING DIRECTORIES

List files in directory:

```
let files = await fs.readdir(".")

console.log(files)
```

Output: ['file1.txt', 'file2.txt', 'folder']

With file types:

```
let entries = await fs.readdir(".", { withFileTypes: true })

entries.forEach(entry => {
```

```
 let type = entry.isDirectory() ? "dir" : "file"
 console.log(`${type}: ${entry.name}`)
})
```

PRACTICE

Read text files with fs.readFile.

Parse JSON files.

Read large files with streams.

Read line by line.

Handle read errors.

WHAT YOU LEARNED

You learned to read files as text or buffer.
You learned sync, callback, and promise patterns.
You learned to read JSON files.
You learned to read line by line.
You learned to handle large files with streams.
You learned to read partial files.
You learned to list directory contents.

The next chapter teaches you about writing files.

WRITING TEXT FILES

Write string to file:

```
const fs = require("fs").promises

await fs.writeFile("output.txt", "Hello, World!")
```

Creates file if it does not exist.
Replaces content if it does exist.

WRITING WITH ENCODING

Specify encoding:

```
await fs.writeFile("output.txt", "Hello!", "utf8")
```

utf8 is default for strings.

WRITING WITH CALLBACKS

```
const fs = require("fs")

fs.writeFile("output.txt", "Hello!", (error) => {
 if (error) {
 console.error(error.message)
 return
 }

 console.log("File written")
})
```

WRITING SYNCHRONOUSLY

```
const fs = require("fs")

fs.writeFileSync("output.txt", "Hello!")
```

Blocks until complete. Use only in scripts.

APPENDING TO FILES

Add to end of file:

```
await fs.appendFile("log.txt", "New line\n")
```

Does not replace. Adds to existing content.

Creates file if it does not exist.

WRITING MULTIPLE LINES

```
let lines = [
 "First line",
 "Second line",
 "Third line"
]

await fs.writeFile("output.txt", lines.join("\n"))
```

Or with newline at end:

```
await fs.writeFile("output.txt", lines.join("\n") + "\n")
```

WRITING JSON

```
let data = {
 name: "Alice",
 age: 25,
 active: true
}

await fs.writeFile("data.json", JSON.stringify(data))
```

Pretty print:

```
await fs.writeFile("data.json", JSON.stringify(data, null, 2))
```

Output:

```
{
 "name": "Alice",
 "age": 25,
 "active": true
}
```

WRITING BUFFERS

Write raw bytes:

```
let buffer = Buffer.from([0x48, 0x65, 0x6c, 0x6c, 0x6f])

await fs.writeFile("binary.dat", buffer)
```

Or from string:

```
let buffer = Buffer.from("Hello")

await fs.writeFile("output.bin", buffer)
```

WRITE OPTIONS

Control write behavior with options:

```
await fs.writeFile("output.txt", "Hello", {
 encoding: "utf8",
 mode: 0o644,
 flag: "w"
})
```

encoding - Text encoding (utf8, ascii, etc.)
mode - File permissions (0o644 is read/write for owner, read for others)
flag - Write mode

FILE FLAGS

w - Write. Create or truncate.
wx - Write. Fail if exists.
a - Append. Create if needed.
ax - Append. Fail if exists.
r+ - Read and write. Fail if not exists.
w+ - Read and write. Create or truncate.
a+ - Read and append. Create if needed.

Use wx to avoid overwriting:

```
try {
 await fs.writeFile("output.txt", "Hello", { flag: "wx" })
} catch (error) {
 if (error.code === "EEXIST") {
 console.log("File already exists")
 }
}
```

ATOMIC WRITES

Write to temp file then rename:

```
const path = require("path")

async function writeAtomic(filePath, content) {
 let tempPath = filePath + ".tmp"

 await fs.writeFile(tempPath, content)
 await fs.rename(tempPath, filePath)
}

await writeAtomic("config.json", JSON.stringify(config))
```

If write fails, original file is not corrupted.

WRITING WITH FILE HANDLES

More control with handles:

```
const fs = require("fs").promises

async function writeWithHandle(path, content) {
 let handle

 try {
 handle = await fs.open(path, "w")
 await handle.writeFile(content)
 } finally {
 if (handle) {
 await handle.close()
 }
```

```
 }
}
```

WRITING AT POSITION

Write at specific position:

```
async function writeAt(path, content, position) {
 let handle = await fs.open(path, "r+")

 try {
 let buffer = Buffer.from(content)
 await handle.write(buffer, 0, buffer.length, position)
 } finally {
 await handle.close()
 }
}

await writeAt("file.txt", "REPLACED", 10)
```

WRITING STREAMS

For large files, use streams:

```
const fs = require("fs")

let stream = fs.createWriteStream("large-output.txt")

for (let i = 0; i < 1000000; i++) {
 stream.write(`Line ${i}\n`)
}

stream.end()
```

```
stream.on("finish", () => {
 console.log("Done writing")
})
```

STREAM WITH BACKPRESSURE

Handle slow writes:

```
const fs = require("fs")

async function writeLines(path, lines) {
 let stream = fs.createWriteStream(path)

 for (let line of lines) {
 let canContinue = stream.write(line + "\n")

 if (!canContinue) {
 await new Promise(resolve => stream.once("drain", resolve))
 }
 }

 stream.end()

 await new Promise(resolve => stream.once("finish", resolve))
}
```

SAFE FILE UPDATES

Pattern for updating files safely:

```
async function updateJson(path, updater) {
 let data

 try {
```

```
 let content = await fs.readFile(path, "utf8")
 data = JSON.parse(content)
 } catch (error) {
 if (error.code === "ENOENT") {
 data = {}
 } else {
 throw error
 }
 }

 let updated = updater(data)

 await fs.writeFile(path, JSON.stringify(updated, null, 2))

 return updated
}

await updateJson("config.json", (config) => {
 return { ...config, lastUpdated: new Date().toISOString() }
})
```

CREATING DIRECTORY BEFORE WRITE

Ensure directory exists:

```
const path = require("path")

async function writeFile(filePath, content) {
 let dir = path.dirname(filePath)

 await fs.mkdir(dir, { recursive: true })
 await fs.writeFile(filePath, content)
}

await writeFile("nested/path/file.txt", "Hello")
```

recursive: true creates all needed directories.

LOGGING TO FILE

Simple file logger:

```
class FileLogger {
 constructor(path) {
 this.path = path
 }

 async log(level, message) {
 let timestamp = new Date().toISOString()
 let line = `[${timestamp}] [${level}] ${message}\n`

 await fs.appendFile(this.path, line)
 }

 info(message) {
 return this.log("INFO", message)
 }

 error(message) {
 return this.log("ERROR", message)
 }
}

let logger = new FileLogger("app.log")

await logger.info("Application started")
await logger.error("Something went wrong")
```

PRACTICE

Write text files.

Append to files.

Write JSON with formatting.

Use atomic writes for important data.

Use streams for large files.

WHAT YOU LEARNED

You learned to write files with fs.writeFile.
You learned to append with fs.appendFile.
You learned file flags for write modes.
You learned atomic write pattern.
You learned write streams for large files.
You learned to handle backpressure.
You learned safe update patterns.

The next chapter teaches you about working with directories.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 11: Working with Directories

CREATING DIRECTORIES

Create a directory:

```
const fs = require("fs").promises

await fs.mkdir("new-folder")
```

Create nested directories:

```
await fs.mkdir("path/to/nested/folder", { recursive: true })
```

recursive: true creates all parent directories if they do not exist.

Without recursive, mkdir fails if parent does not exist.

CHECKING IF DIRECTORY EXISTS

```
async function directoryExists(path) {
 try {
 let stats = await fs.stat(path)
 return stats.isDirectory()
 } catch {
 return false
 }
}

if (await directoryExists("my-folder")) {
 console.log("Directory exists")
}
```

READING DIRECTORY CONTENTS

List files and folders:

```
let items = await fs.readdir(".")

console.log(items)
```

Output: ['file1.txt', 'file2.txt', 'subfolder']

With file type information:

```
let items = await fs.readdir(".", { withFileTypes: true })
```

```
items.forEach(item => {
 if (item.isDirectory()) {
 console.log(`[DIR] ${item.name}`)
 } else {
 console.log(`[FILE] ${item.name}`)
 }
})
```

FILTERING BY TYPE

Get only files:

```
async function getFiles(directory) {
 let items = await fs.readdir(directory, { withFileTypes: true })

 return items
 .filter(item => item.isFile())
 .map(item => item.name)
}
```

Get only directories:

```
async function getDirectories(directory) {
 let items = await fs.readdir(directory, { withFileTypes: true })

 return items
 .filter(item => item.isDirectory())
 .map(item => item.name)
}
```

FILTERING BY EXTENSION

```
async function getFilesByExtension(directory, extension) {
```

```
 let items = await fs.readdir(directory)

 return items.filter(item => item.endsWith(extension))
}

let jsFiles = await getFilesByExtension(".", ".js")

console.log(jsFiles)
```

RECURSIVE DIRECTORY LISTING

List all files in all subdirectories:

```
const path = require("path")

async function getAllFiles(directory) {
 let files = []
 let items = await fs.readdir(directory, { withFileTypes: true })

 for (let item of items) {
 let fullPath = path.join(directory, item.name)

 if (item.isDirectory()) {
 let subFiles = await getAllFiles(fullPath)
 files.push(...subFiles)
 } else {
 files.push(fullPath)
 }
 }

 return files
}

let allFiles = await getAllFiles(".")

console.log(allFiles)
```

RECURSIVE WITH NODE 20+

Node.js 20+ has recursive option:

let files = await fs.readdir(".", { recursive: true })

console.log(files)

Returns all files and folders recursively.

DELETING DIRECTORIES

Delete empty directory:

await fs.rmdir("empty-folder")

Fails if directory is not empty.

Delete directory with contents:

await fs.rm("folder-with-files", { recursive: true })

recursive: true removes all contents.

Force delete (no error if missing):

await fs.rm("maybe-exists", { recursive: true, force: true })

MOVING DIRECTORIES

Use rename to move:

```
await fs.rename("old-location/folder", "new-location/folder")
```

This works if source and destination are on the same filesystem.

COPYING DIRECTORIES

Node.js 16.7+ has recursive copy:

```
await fs.cp("source-folder", "dest-folder", { recursive: true })
```

For older versions, copy manually:

```
async function copyDir(source, dest) {
 await fs.mkdir(dest, { recursive: true })

 let items = await fs.readdir(source, { withFileTypes: true })

 for (let item of items) {
 let srcPath = path.join(source, item.name)
 let destPath = path.join(dest, item.name)

 if (item.isDirectory()) {
 await copyDir(srcPath, destPath)
 } else {
 await fs.copyFile(srcPath, destPath)
 }
 }
}

await copyDir("source", "destination")
```

WATCHING DIRECTORIES

Watch for changes:

```
fs.watch(".", (eventType, filename) => {
 console.log(`${filename}: ${eventType}`)
})
```

Watch recursively (Node.js 18+):

```
fs.watch(".", { recursive: true }, (eventType, filename) => {
 console.log(`${filename}: ${eventType}`)
})
```

FINDING FILES

Find files matching a pattern:

```
async function findFiles(directory, pattern) {
 let matches = []
 let regex = new RegExp(pattern)
 let items = await fs.readdir(directory, { withFileTypes: true })

 for (let item of items) {
 let fullPath = path.join(directory, item.name)

 if (item.isDirectory()) {
 let subMatches = await findFiles(fullPath, pattern)
 matches.push(...subMatches)
 } else if (regex.test(item.name)) {
 matches.push(fullPath)
 }
 }

 return matches
}

let testFiles = await findFiles(".", "\\.test\\.js$")
```

```
console.log(testFiles)
```

DIRECTORY SIZE

Calculate total size:

```
async function getDirectorySize(directory) {
 let totalSize = 0
 let items = await fs.readdir(directory, { withFileTypes: true })

 for (let item of items) {
 let fullPath = path.join(directory, item.name)

 if (item.isDirectory()) {
 totalSize += await getDirectorySize(fullPath)
 } else {
 let stats = await fs.stat(fullPath)
 totalSize += stats.size
 }
 }

 return totalSize
}

let size = await getDirectorySize(".")

console.log(`Size: ${(size / 1024 / 1024).toFixed(2)} MB`)
```

CREATING TEMP DIRECTORIES

```
const os = require("os")

async function createTempDir(prefix = "temp-") {
 let tempBase = os.tmpdir()
```

```
 let tempPath = path.join(tempBase, `${prefix}${Date.now()}`)

 await fs.mkdir(tempPath)

 return tempPath
}

let tempDir = await createTempDir()

console.log("Temp dir:", tempDir)
```

Or use mkdtemp:

```
let tempDir = await fs.mkdtemp(path.join(os.tmpdir(), "my-app-"))
```

ENSURING DIRECTORY

Create if not exists:

```
async function ensureDir(directory) {
 try {
 await fs.access(directory)
 } catch {
 await fs.mkdir(directory, { recursive: true })
 }
}

await ensureDir("data/cache")
```

EMPTY DIRECTORY

Remove all contents but keep directory:

```
async function emptyDir(directory) {
```

```
 let items = await fs.readdir(directory)

 for (let item of items) {
 await fs.rm(path.join(directory, item), { recursive: true })
 }
}

await emptyDir("temp")
```

PRACTICE

Create and delete directories.

List directory contents recursively.

Filter files by extension.

Copy and move directories.

Calculate directory size.

WHAT YOU LEARNED

You learned to create directories with mkdir.
You learned to read directory contents.
You learned recursive directory operations.
You learned to delete directories with rm.
You learned to copy and move directories.
You learned to watch for changes.
You learned directory utility functions.

The next chapter teaches you about paths.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 12: Paths

THE PATH MODULE

The path module handles file and directory paths.

```
const path = require("path")
```

It handles differences between operating systems:
- Windows uses backslashes: C:\Users\name
- Linux/macOS uses forward slashes: /home/name

The path module works correctly on any system.

PATH.JOIN

Join path segments:

```
let fullPath = path.join("folder", "subfolder", "file.txt")

console.log(fullPath)
```

On Linux: folder/subfolder/file.txt
On Windows: folder\subfolder\file.txt

Handles extra slashes:

```
path.join("folder/", "/subfolder/", "/file.txt")
```

Result: folder/subfolder/file.txt

Always use path.join instead of string concatenation.

Bad:

let p = folder + "/" + file

Good:

let p = path.join(folder, file)

PATH.RESOLVE

Get absolute path:

let absolute = path.resolve("folder", "file.txt")

console.log(absolute)

Output: /home/user/project/folder/file.txt

resolve starts from current directory and builds absolute path.

With absolute segment, starts there:

path.resolve("/tmp", "file.txt")

Output: /tmp/file.txt

PATH.DIRNAME

Get directory of a path:

let dir = path.dirname("/home/user/file.txt")

console.log(dir)

Output: /home/user

Useful for finding containing folder:

let scriptDir = path.dirname(__filename)

PATH.BASENAME

Get file name from path:

let name = path.basename("/home/user/file.txt")

console.log(name)

Output: file.txt

Remove extension:

let name = path.basename("/home/user/file.txt", ".txt")

console.log(name)

Output: file

PATH.EXTNAME

Get file extension:

let ext = path.extname("file.txt")

console.log(ext)

Output: .txt

Includes the dot.

No extension returns empty string:

path.extname("README")

Output: ""

PATH.PARSE

Parse path into components:

let parsed = path.parse("/home/user/project/file.txt")

console.log(parsed)

Output:
{
 root: "/",
 dir: "/home/user/project",
 base: "file.txt",
 ext: ".txt",
 name: "file"
}

root - Root portion (/ or C:\)
dir - Directory portion
base - Filename with extension
ext - Extension with dot
name - Filename without extension

PATH.FORMAT

Build path from components:

```
let pathStr = path.format({
 dir: "/home/user",
 name: "file",
 ext: ".txt"
})

console.log(pathStr)
```

Output: /home/user/file.txt

Opposite of path.parse.

PATH.NORMALIZE

Clean up messy paths:

```
let clean = path.normalize("/home/user/../other/./file.txt")

console.log(clean)
```

Output: /home/other/file.txt

Resolves . and .. and fixes slashes.

PATH.RELATIVE

Get relative path between two paths:

```
let rel = path.relative("/home/user/project", "/home/user/other/
file.txt")

console.log(rel)
```

Output: ../other/file.txt

Useful for displaying shorter paths.

PATH.ISABSOLUTE

Check if path is absolute:

path.isAbsolute("/home/user")

Output: true

path.isAbsolute("relative/path")

Output: false

On Windows:

path.isAbsolute("C:\\Users")

Output: true

PATH.SEP

Platform-specific separator:

console.log(path.sep)

On Linux/macOS: /
On Windows: \

Useful for splitting paths:

let parts = filePath.split(path.sep)

PATH.DELIMITER

Platform-specific path delimiter:

console.log(path.delimiter)

On Linux/macOS: :
On Windows: ;

Used in PATH environment variable:

let paths = process.env.PATH.split(path.delimiter)

WINDOWS PATHS ON ANY SYSTEM

path.win32 forces Windows behavior:

path.win32.join("folder", "file")

Output: folder\file

path.posix forces Unix behavior:

path.posix.join("folder", "file")

Output: folder/file

Useful for cross-platform compatibility.

COMMON PATTERNS

Get directory of current file:

```
let thisDir = path.dirname(_filename)
```

Resolve relative to current file:

```
let configPath = path.resolve(_dirname, "config.json")
```

Get file extension:

```
function hasExtension(filePath, ext) {
 return path.extname(filePath).toLowerCase() = = =
ext.toLowerCase()
}

if (hasExtension(file, ".json")) {
 parseJson(file)
}
```

Change extension:

```
function changeExtension(filePath, newExt) {
 let parsed = path.parse(filePath)
 return path.join(parsed.dir, parsed.name + newExt)
}

changeExtension("file.txt", ".md")
```

Output: file.md

Get unique filename:

```
function uniquePath(filePath) {
 let parsed = path.parse(filePath)
 let timestamp = Date.now()
 return path.join(parsed.dir, `${parsed.name}-
${timestamp}${parsed.ext}`)
}
```

PATH IN ES MODULES

ES modules use import.meta.url instead of _filename:

```
import { fileURLToPath } from "url"
import { dirname, join } from "path"

const _filename = fileURLToPath(import.meta.url)
const _dirname = dirname(_filename)

let configPath = join(_dirname, "config.json")
```

VALIDATING PATHS

Check for directory traversal attacks:

```
function isSafePath(basePath, requestedPath) {
 let full = path.resolve(basePath, requestedPath)
 let normalized = path.normalize(full)

 return normalized.startsWith(basePath)
}

let base = "/var/www/uploads"
let requested = "../../../etc/passwd"

if (isSafePath(base, requested)) {
 readFile(path.join(base, requested))
} else {
 console.log("Access denied")
}
```

This prevents users from escaping allowed directories.

PRACTICE

Use path.join for building paths.

Use path.resolve for absolute paths.

Parse and format paths.

Handle cross-platform differences.

Validate paths for security.

WHAT YOU LEARNED

You learned path.join for combining paths.
You learned path.resolve for absolute paths.
You learned path.dirname, basename, extname.
You learned path.parse and format.
You learned cross-platform path handling.
You learned path validation for security.

The next chapter teaches you about streams.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 13: Streams

WHAT ARE STREAMS

Streams process data piece by piece.

Instead of loading everything into memory, streams handle data in chunks.

Reading a 1GB file:

Without streams:

let data = fs.readFileSync("huge.txt")

Loads entire file into memory. 1GB of RAM used.

With streams:

let stream = fs.createReadStream("huge.txt")

Processes in chunks. Maybe 64KB at a time. Memory stays low.

Streams are essential for large files and real-time data.

TYPES OF STREAMS

Readable - Data source you read from
Writable - Data destination you write to
Duplex - Both readable and writable
Transform - Duplex that modifies data

Examples:

Readable: fs.createReadStream, http request
Writable: fs.createWriteStream, http response
Duplex: net.Socket, crypto streams
Transform: zlib compression, encryption

READABLE STREAMS

Create readable stream:

```
const fs = require("fs")

let stream = fs.createReadStream("file.txt", {
 encoding: "utf8",
 highWaterMark: 64 * 1024
})
```

highWaterMark is chunk size in bytes. Default is 64KB.

Read with events:

```
stream.on("data", (chunk) => {
 console.log("Received:", chunk.length, "bytes")
})

stream.on("end", () => {
 console.log("Done reading")
})

stream.on("error", (error) => {
 console.error("Error:", error.message)
})
```

PAUSE AND RESUME

Control flow:

```
stream.pause()

stream.resume()
```

Useful when you need time to process each chunk.

```
stream.on("data", async (chunk) => {
 stream.pause()
 await processChunk(chunk)
```

```
 stream.resume()
})
```

WRITABLE STREAMS

Create writable stream:

```
let stream = fs.createWriteStream("output.txt")

stream.write("First chunk\n")
stream.write("Second chunk\n")
stream.write("Third chunk\n")

stream.end()

stream.on("finish", () => {
 console.log("Done writing")
})

stream.on("error", (error) => {
 console.error("Error:", error.message)
})
```

BACKPRESSURE

write returns false when internal buffer is full:

```
let canContinue = stream.write(data)

if (!canContinue) {
 stream.once("drain", () => {
 continueWriting()
 })
}
```

Respect backpressure to avoid memory issues:

```
async function writeData(stream, chunks) {
 for (let chunk of chunks) {
 let canContinue = stream.write(chunk)

 if (!canContinue) {
 await new Promise(resolve => stream.once("drain", resolve))
 }
 }

 stream.end()
}
```

PIPING STREAMS

Connect readable to writable:

```
let readable = fs.createReadStream("input.txt")
let writable = fs.createWriteStream("output.txt")

readable.pipe(writable)
```

Pipe handles backpressure automatically.

Chain multiple pipes:

```
readable
 .pipe(transform1)
 .pipe(transform2)
 .pipe(writable)
```

PIPE WITH ERROR HANDLING

```
const { pipeline } = require("stream")

pipeline(
 fs.createReadStream("input.txt"),
 transformStream,
 fs.createWriteStream("output.txt"),
 (error) => {
 if (error) {
 console.error("Pipeline failed:", error)
 } else {
 console.log("Pipeline succeeded")
 }
 }
)
```

Or with promises:

```
const { pipeline } = require("stream/promises")

await pipeline(
 fs.createReadStream("input.txt"),
 transformStream,
 fs.createWriteStream("output.txt")
)
```

TRANSFORM STREAMS

Transform modifies data as it flows:

```
const { Transform } = require("stream")

let upperCase = new Transform({
 transform(chunk, encoding, callback) {
 let upper = chunk.toString().toUpperCase()
 callback(null, upper)
 }
```

```
})

fs.createReadStream("input.txt")
 .pipe(upperCase)
 .pipe(fs.createWriteStream("output.txt"))
```

CREATING CUSTOM TRANSFORM

```
const { Transform } = require("stream")

class LineCounter extends Transform {
 constructor() {
 super()
 this.count = 0
 }

 _transform(chunk, encoding, callback) {
 let text = chunk.toString()
 let lines = text.split("\n").length - 1

 this.count += lines
 this.push(chunk)

 callback()
 }

 _flush(callback) {
 this.push(`\nTotal lines: ${this.count}\n`)
 callback()
 }
}

let counter = new LineCounter()

fs.createReadStream("file.txt")
 .pipe(counter)
 .pipe(process.stdout)
```

DUPLEX STREAMS

Both readable and writable:

```
const { Duplex } = require("stream")

class Echo extends Duplex {
 constructor() {
 super()
 this.data = []
 }

 _write(chunk, encoding, callback) {
 this.data.push(chunk)
 callback()
 }

 _read() {
 if (this.data.length > 0) {
 this.push(this.data.shift())
 } else {
 this.push(null)
 }
 }
}
```

STREAM FROM ITERABLE

Create readable from array or generator:

```
const { Readable } = require("stream")

let stream = Readable.from(["one", "two", "three"])
```

```
stream.on("data", (chunk) => {
 console.log(chunk)
})
```

From generator:

```
async function* generateData() {
 for (let i = 0; i < 5; i++) {
 yield `Line ${i}\n`
 }
}

let stream = Readable.from(generateData())
```

ASYNC ITERATION

Read streams with for-await:

```
async function processStream(stream) {
 for await (let chunk of stream) {
 console.log("Chunk:", chunk.length)
 }
}

let readable = fs.createReadStream("file.txt")

await processStream(readable)
```

COMPRESSION WITH STREAMS

```
const zlib = require("zlib")

fs.createReadStream("file.txt")
```

```
 .pipe(zlib.createGzip())
 .pipe(fs.createWriteStream("file.txt.gz"))
```

Decompress:

```
fs.createReadStream("file.txt.gz")
 .pipe(zlib.createGunzip())
 .pipe(fs.createWriteStream("file.txt"))
```

STREAM TO STRING

Collect stream into string:

```
async function streamToString(stream) {
 let chunks = []

 for await (let chunk of stream) {
 chunks.push(chunk)
 }

 return Buffer.concat(chunks).toString("utf8")
}

let content = await streamToString(fs.createReadStream("file.txt"))
```

PRACTICE

Use readable and writable streams.

Pipe streams together.

Handle backpressure correctly.

Create transform streams.

Use async iteration with streams.

WHAT YOU LEARNED

You learned streams process data in chunks.
You learned readable, writable, duplex, transform types.
You learned to pipe streams together.
You learned backpressure handling.
You learned to create transform streams.
You learned async iteration over streams.

The next chapter teaches you about events.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 14: Events

EVENT-DRIVEN PROGRAMMING

Node.js is built on events.

Instead of calling functions directly, you emit events and listen for them.

This creates loose coupling. The emitter does not know who is listening. Listeners do not know who is emitting.

Many Node.js core modules use events: http, fs, streams, process.

THE EVENTS MODULE

```
const { EventEmitter } = require("events")

let emitter = new EventEmitter()
```

EventEmitter is the base class for event-based objects.

LISTENING FOR EVENTS

Register event listeners with on:

```
emitter.on("message", (data) => {
 console.log("Received:", data)
})
```

The first argument is event name.
The second is the handler function.

EMITTING EVENTS

Trigger events with emit:

```
emitter.emit("message", "Hello!")
```

Output: Received: Hello!

First argument is event name.
Additional arguments pass to handlers.

```
emitter.emit("message", "Hello", { id: 1 })

emitter.on("message", (text, info) => {
 console.log(text, info)
})
```

MULTIPLE LISTENERS

Add multiple listeners to same event:

```
emitter.on("data", (d) => console.log("Listener 1:", d))
emitter.on("data", (d) => console.log("Listener 2:", d))
emitter.on("data", (d) => console.log("Listener 3:", d))

emitter.emit("data", "test")
```

Output:
Listener 1: test
Listener 2: test
Listener 3: test

Listeners run in order they were added.

ONCE

Listen for event only once:

```
emitter.once("connect", () => {
 console.log("Connected!")
})

emitter.emit("connect")
emitter.emit("connect")
```

Output: Connected!

Second emit does nothing. Handler was removed after first call.

REMOVING LISTENERS

Remove specific listener:

```
function handler(data) {
 console.log(data)
}

emitter.on("data", handler)

emitter.off("data", handler)
```

Must pass same function reference.

Remove all listeners for event:

```
emitter.removeAllListeners("data")
```

Remove all listeners:

```
emitter.removeAllListeners()
```

PREPENDING LISTENERS

Add listener to beginning:

```
emitter.prependListener("data", handler)

emitter.prependOnceListener("data", handler)
```

These listeners run before previously added ones.

GETTING LISTENER INFO

Count listeners:

let count = emitter.listenerCount("data")

Get listener functions:

let listeners = emitter.listeners("data")

Get event names:

let events = emitter.eventNames()

MAX LISTENERS

Default limit is 10 listeners per event.

Going over warns about memory leaks.

Change the limit:

emitter.setMaxListeners(20)

Or unlimited:

emitter.setMaxListeners(0)

ERROR EVENTS

error is a special event.

If no listener and error is emitted, Node.js crashes.

emitter.emit("error", new Error("Something broke"))

Always handle error events:

```
emitter.on("error", (error) => {
 console.error("Error:", error.message)
})
```

CREATING EVENT-BASED CLASSES

Extend EventEmitter:

```
const { EventEmitter } = require("events")

class Database extends EventEmitter {
 connect() {
 setTimeout(() => {
 this.emit("connected")
 }, 1000)
 }

 query(sql) {
 setTimeout(() => {
 this.emit("result", { rows: [] })
 }, 500)
 }

 close() {
 this.emit("close")
 }
}

let db = new Database()

db.on("connected", () => console.log("Database connected"))
db.on("result", (data) => console.log("Query result:", data))
db.on("close", () => console.log("Connection closed"))
```

db.connect()

ASYNC EVENT HANDLERS

Handlers can be async:

```
emitter.on("data", async (data) = > {
 await processData(data)
 console.log("Done processing")
})
```

Warning: Errors in async handlers are unhandled rejections.

Handle errors:

```
emitter.on("data", async (data) = > {
 try {
 await processData(data)
 } catch (error) {
 emitter.emit("error", error)
 }
})
```

WAITING FOR EVENTS

Use events.once to promisify:

```
const { once } = require("events")

async function waitForConnect(emitter) {
 await once(emitter, "connect")
 console.log("Connected!")
}
```

With abort:

```
const { once } = require("events")

let ac = new AbortController()

setTimeout(() => ac.abort(), 5000)

try {
 await once(emitter, "data", { signal: ac.signal })
} catch (error) {
 if (error.name === "AbortError") {
 console.log("Timed out")
 }
}
```

EVENT EMITTER AS ASYNC ITERATOR

```
const { on } = require("events")

async function main() {
 let ac = new AbortController()

 setTimeout(() => ac.abort(), 10000)

 try {
 for await (let [data] of on(emitter, "data", { signal: ac.signal })) {
 console.log("Received:", data)
 }
 } catch (error) {
 if (error.name === "AbortError") {
 console.log("Stopped listening")
 }
 }
}
```

REAL-WORLD EXAMPLE

Job queue with events:

```
class JobQueue extends EventEmitter {
 constructor() {
 super()
 this.jobs = []
 this.processing = false
 }

 add(job) {
 this.jobs.push(job)
 this.emit("job:added", job)
 this.process()
 }

 async process() {
 if (this.processing || this.jobs.length === 0) {
 return
 }

 this.processing = true

 while (this.jobs.length > 0) {
 let job = this.jobs.shift()

 this.emit("job:started", job)

 try {
 let result = await this.executeJob(job)
 this.emit("job:completed", job, result)
 } catch (error) {
 this.emit("job:failed", job, error)
 }
 }

 this.processing = false
```

```
 this.emit("queue:empty")
 }

 async executeJob(job) {
 return new Promise(resolve => {
 setTimeout(() => resolve(`Done: ${job}`), 1000)
 })
 }
}

let queue = new JobQueue()

queue.on("job:started", (job) => console.log(`Starting: ${job}`))
queue.on("job:completed", (job, result) =>
console.log(`Completed: ${result}`))
queue.on("queue:empty", () => console.log("All jobs done"))

queue.add("Task 1")
queue.add("Task 2")
queue.add("Task 3")
```

PRACTICE

Create EventEmitter instances.

Listen with on and once.

Emit events with data.

Extend EventEmitter for custom classes.

Handle error events.

WHAT YOU LEARNED

You learned EventEmitter basics.
You learned to listen and emit events.
You learned once for single-time listeners.
You learned to remove listeners.
You learned error event handling.
You learned to extend EventEmitter.
You learned async iteration over events.

The next chapter teaches you about the HTTP module.

BOOK 13: NODE.JS FUNDAMENTALS

Chapter 15: HTTP Module

THE HTTP MODULE

Node.js can create HTTP servers and make HTTP requests.

```
const http = require("http")
```

No external libraries needed for basic HTTP.

CREATING A SIMPLE SERVER

```
const http = require("http")

let server = http.createServer((request, response) => {
 response.writeHead(200, { "Content-Type": "text/plain" })
 response.end("Hello, World!")
})

server.listen(3000, () => {
 console.log("Server running at http://localhost:3000/")
})
```

createServer takes a callback that runs for every request.
request contains information about the incoming request.
response is used to send data back.

REQUEST OBJECT

The request object contains:

request.method - HTTP method (GET, POST, etc.)
request.url - Request URL
request.headers - Request headers

```
let server = http.createServer((req, res) => {
 console.log("Method:", req.method)
 console.log("URL:", req.url)
 console.log("Headers:", req.headers)

 res.end("OK")
})
```

RESPONSE OBJECT

The response object has methods:

response.writeHead(statusCode, headers) - Set status and headers
response.setHeader(name, value) - Set single header
response.write(data) - Write to body
response.end(data) - End response, optionally with final data

```
res.writeHead(200, {
 "Content-Type": "text/html",
 "X-Custom-Header": "value"
})
```

```
res.write("<h1>Hello</h1>")
res.write("<p>World</p>")
res.end()
```

STATUS CODES

Set status code:

```
res.statusCode = 404
res.end("Not Found")
```

Or with writeHead:

```
res.writeHead(404, { "Content-Type": "text/plain" })
res.end("Not Found")
```

Common codes:

200 - OK
201 - Created
301 - Moved Permanently
302 - Found (redirect)
400 - Bad Request
401 - Unauthorized
403 - Forbidden
404 - Not Found
500 - Internal Server Error

CONTENT TYPES

Set appropriate Content-Type:

HTML:

res.setHeader("Content-Type", "text/html")

Plain text:
res.setHeader("Content-Type", "text/plain")

JSON:
res.setHeader("Content-Type", "application/json")

CSS:
res.setHeader("Content-Type", "text/css")

JavaScript:
res.setHeader("Content-Type", "application/javascript")

RESPONDING WITH JSON

```
let server = http.createServer((req, res) => {
 let data = {
 message: "Hello",
 timestamp: Date.now()
 }

 res.writeHead(200, { "Content-Type": "application/json" })
 res.end(JSON.stringify(data))
})
```

SIMPLE ROUTING

```
let server = http.createServer((req, res) => {
 if (req.url === "/") {
 res.writeHead(200, { "Content-Type": "text/html" })
 res.end("<h1>Home Page</h1>")
 } else if (req.url === "/about") {
 res.writeHead(200, { "Content-Type": "text/html" })
```

```
 res.end("<h1>About Page</h1>")
 } else if (req.url === "/api/data") {
 res.writeHead(200, { "Content-Type": "application/json" })
 res.end(JSON.stringify({ data: "test" }))
 } else {
 res.writeHead(404, { "Content-Type": "text/html" })
 res.end("<h1>404 Not Found</h1>")
 }
})
```

ROUTING WITH METHOD

```
let server = http.createServer((req, res) => {
 if (req.method === "GET" && req.url === "/users") {
 res.writeHead(200, { "Content-Type": "application/json" })
 res.end(JSON.stringify([{ id: 1, name: "Alice" }]))
 } else if (req.method === "POST" && req.url === "/users") {
 let body = ""

 req.on("data", chunk => body += chunk)

 req.on("end", () => {
 let user = JSON.parse(body)
 res.writeHead(201, { "Content-Type": "application/json" })
 res.end(JSON.stringify({ id: 2, ...user }))
 })
 } else {
 res.writeHead(404)
 res.end("Not Found")
 }
})
```

PARSING REQUEST BODY

Request body comes as a stream:

```
let server = http.createServer((req, res) => {
 if (req.method === "POST") {
 let body = ""

 req.on("data", (chunk) => {
 body += chunk
 })

 req.on("end", () => {
 console.log("Received:", body)

 let data = JSON.parse(body)

 res.writeHead(200, { "Content-Type": "application/json" })
 res.end(JSON.stringify({ received: data }))
 })
 }
})
```

Helper function:

```
function getBody(req) {
 return new Promise((resolve, reject) => {
 let body = ""

 req.on("data", chunk => body += chunk)
 req.on("end", () => resolve(body))
 req.on("error", reject)
 })
}

let body = await getBody(req)
```

PARSING URL AND QUERY STRING

```
const url = require("url")

let server = http.createServer((req, res) => {
 let parsed = new URL(req.url, `http://${req.headers.host}`)

 console.log("Pathname:", parsed.pathname)
 console.log("Query:", parsed.searchParams)

 let name = parsed.searchParams.get("name")

 res.end(`Hello, ${name ?? "World"}!`)
})
```

Request: http://localhost:3000/greet?name=Alice

Output: Hello, Alice!

SERVER EVENTS

```
let server = http.createServer()

server.on("request", (req, res) => {
 res.end("Hello")
})

server.on("listening", () => {
 console.log("Server started")
})

server.on("error", (error) => {
 console.error("Server error:", error)
})

server.on("close", () => {
 console.log("Server closed")
})
```

```
server.listen(3000)
```

STOPPING THE SERVER

```
server.close(() => {
 console.log("Server stopped")
})
```

Current connections continue until complete.

MAKING HTTP REQUESTS

http.request makes outgoing requests:

```
const http = require("http")

let options = {
 hostname: "api.example.com",
 port: 80,
 path: "/data",
 method: "GET"
}

let req = http.request(options, (res) => {
 let data = ""

 res.on("data", chunk => data += chunk)

 res.on("end", () => {
 console.log("Response:", data)
 })
})

req.on("error", (error) => {
```

```
 console.error("Request error:", error)
})

req.end()
```

For HTTPS, use require("https").

SIMPLER GET REQUESTS

http.get for simple GET:

```
http.get("http://api.example.com/data", (res) => {
 let data = ""

 res.on("data", chunk => data += chunk)

 res.on("end", () => {
 console.log(data)
 })
}).on("error", (error) => {
 console.error(error)
})
```

PRACTICE

Create HTTP servers.

Handle different routes and methods.

Parse request body.

Set response headers and status codes.

Make HTTP requests.

WHAT YOU LEARNED

You learned to create HTTP servers.
You learned request and response objects.
You learned basic routing.
You learned to parse request bodies.
You learned to parse URLs and query strings.
You learned to make outgoing requests.

The next chapter goes deeper into creating servers.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 16: Creating a Server

STRUCTURED SERVER

Let us build a proper HTTP server step by step.

```
const http = require("http")

class Server {
 constructor() {
 this.server = null
 this.port = 3000
 }

 start() {
 this.server = http.createServer((req, res) => {
 this.handleRequest(req, res)
 })

 this.server.listen(this.port, () => {
 console.log(`Server running on port ${this.port}`)
```

```
 })

 this.server.on("error", (error) => {
 console.error("Server error:", error.message)
 })
 }

 handleRequest(req, res) {
 res.writeHead(200, { "Content-Type": "text/plain" })
 res.end("Hello from server")
 }

 stop() {
 if (this.server) {
 this.server.close()
 }
 }
}

let app = new Server()

app.start()
```

ROUTER CLASS

```
class Router {
 constructor() {
 this.routes = []
 }

 get(path, handler) {
 this.routes.push({ method: "GET", path, handler })
 }

 post(path, handler) {
 this.routes.push({ method: "POST", path, handler })
 }
```

```
put(path, handler) {
this.routes.push({ method: "PUT", path, handler })
}

delete(path, handler) {
this.routes.push({ method: "DELETE", path, handler })
}

match(method, url) {
let pathname = new URL(url, "http://localhost").pathname

for (let route of this.routes) {
if (route.method !== method) continue

let params = this.matchPath(route.path, pathname)

if (params !== null) {
return { handler: route.handler, params }
}
}

return null
}

matchPath(pattern, pathname) {
let patternParts = pattern.split("/")
let pathParts = pathname.split("/")

if (patternParts.length !== pathParts.length) {
return null
}

let params = {}

for (let i = 0; i < patternParts.length; i++) {
if (patternParts[i].startsWith(":")) {
let paramName = patternParts[i].slice(1)
params[paramName] = pathParts[i]
} else if (patternParts[i] !== pathParts[i]) {
```

```
 return null
 }
 }

 return params
 }
}
```

USING THE ROUTER

```
let router = new Router()

router.get("/", (req, res) => {
 res.json({ message: "Home" })
})

router.get("/users", (req, res) => {
 res.json([{ id: 1, name: "Alice" }])
})

router.get("/users/:id", (req, res) => {
 res.json({ id: req.params.id })
})

router.post("/users", async (req, res) => {
 let body = await req.body
 res.json({ created: body })
})
```

ENHANCED REQUEST

Add helpers to request:

```
function enhanceRequest(req) {
```

```
 let url = new URL(req.url, `http://${req.headers.host}`)

 req.pathname = url.pathname
 req.query = Object.fromEntries(url.searchParams)
 req.params = {}

 req.body = new Promise((resolve, reject) => {
 let data = ""

 req.on("data", chunk => data += chunk)
 req.on("end", () => {
 try {
 resolve(data ? JSON.parse(data) : null)
 } catch {
 resolve(data)
 }
 })
 req.on("error", reject)
 })

 return req
}
```

ENHANCED RESPONSE

Add helpers to response:

```
function enhanceResponse(res) {
 res.json = (data) => {
 res.setHeader("Content-Type", "application/json")
 res.end(JSON.stringify(data))
 }

 res.html = (content) => {
 res.setHeader("Content-Type", "text/html")
 res.end(content)
 }
```

```
 res.send = (data) => {
 if (typeof data === "object") {
 res.json(data)
 } else {
 res.setHeader("Content-Type", "text/plain")
 res.end(String(data))
 }
 }

 res.status = (code) => {
 res.statusCode = code
 return res
 }

 res.redirect = (url, code = 302) => {
 res.writeHead(code, { Location: url })
 res.end()
 }

 return res
}
```

COMPLETE SERVER CLASS

```
const http = require("http")

class Server {
 constructor() {
 this.server = null
 this.router = new Router()
 this.middlewares = []
 }

 use(middleware) {
 this.middlewares.push(middleware)
 }
```

```
get(path, handler) {
this.router.get(path, handler)
}

post(path, handler) {
this.router.post(path, handler)
}

put(path, handler) {
this.router.put(path, handler)
}

delete(path, handler) {
this.router.delete(path, handler)
}

async handleRequest(req, res) {
enhanceRequest(req)
enhanceResponse(res)

for (let middleware of this.middlewares) {
let shouldContinue = await middleware(req, res)

if (shouldContinue === false || res.writableEnded) {
return
}
}

let match = this.router.match(req.method, req.url)

if (match) {
req.params = match.params

try {
await match.handler(req, res)
} catch (error) {
console.error("Handler error:", error)
res.status(500).json({ error: "Internal server error" })
}
```

```
 } else {
 res.status(404).json({ error: "Not found" })
 }
 }

 listen(port, callback) {
 this.server = http.createServer((req, res) => {
 this.handleRequest(req, res)
 })

 this.server.listen(port, callback)
 }

 close() {
 if (this.server) {
 this.server.close()
 }
 }
}
```

USING THE SERVER

```
let app = new Server()

app.use((req, res) => {
 console.log(`${req.method} ${req.url}`)
})

app.get("/", (req, res) => {
 res.json({ message: "Welcome to the API" })
})

app.get("/users", (req, res) => {
 let users = [
 { id: 1, name: "Alice" },
 { id: 2, name: "Bob" }
 ]
```

```
 res.json(users)
})

app.get("/users/:id", (req, res) => {
 res.json({ id: req.params.id })
})

app.post("/users", async (req, res) => {
 let body = await req.body

 res.status(201).json({
 id: Date.now(),
 ...body
 })
})

app.listen(3000, () => {
 console.log("Server running on http://localhost:3000")
})
```

MIDDLEWARE EXAMPLES

Logger:

```
app.use((req, res) => {
 let start = Date.now()

 res.on("finish", () => {
 let duration = Date.now() - start
 console.log(`${req.method} ${req.url} - ${res.statusCode} -
${duration}ms`)
 })
})
```

CORS:

```
app.use((req, res) = > {
 res.setHeader("Access-Control-Allow-Origin", "*")
 res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT,
DELETE")
 res.setHeader("Access-Control-Allow-Headers", "Content-Type")

 if (req.method = = = "OPTIONS") {
 res.status(204).end()
 return false
 }
})
```

Authentication:

```
app.use((req, res) = > {
 let token = req.headers.authorization?.replace("Bearer ", "")

 if (token) {
 req.user = verifyToken(token)
 }
})
```

ERROR HANDLING MIDDLEWARE

```
app.use(async (req, res) = > {
 try {
 return true
 } catch (error) {
 console.error("Middleware error:", error)
 res.status(500).json({ error: "Server error" })
 return false
 }
})
```

PRACTICE

Build server with router.

Add request and response helpers.

Implement middleware support.

Handle route parameters.

Handle errors properly.

WHAT YOU LEARNED

You learned to structure a server class.
You learned to build a router with parameters.
You learned to enhance request and response.
You learned middleware pattern.
You learned error handling.

The next chapter teaches you about handling requests.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 17: Handling Requests

REQUEST METHODS

Handle different HTTP methods:

```
let server = http.createServer(async (req, res) => {
 switch (req.method) {
 case "GET":
 handleGet(req, res)
 break
```

```
 case "POST":
 await handlePost(req, res)
 break
 case "PUT":
 await handlePut(req, res)
 break
 case "PATCH":
 await handlePatch(req, res)
 break
 case "DELETE":
 handleDelete(req, res)
 break
 case "OPTIONS":
 handleOptions(req, res)
 break
 default:
 res.writeHead(405)
 res.end("Method Not Allowed")
 }
})
```

PARSING JSON BODY

```
function parseJsonBody(req) {
 return new Promise((resolve, reject) = > {
 let body = ""

 req.on("data", chunk = > body + = chunk)

 req.on("end", () = > {
 if (!body) {
 resolve(null)
 return
 }

 try {
 resolve(JSON.parse(body))
```

```
 } catch (error) {
 reject(new Error("Invalid JSON"))
 }
 })

 req.on("error", reject)
 })
}

async function handlePost(req, res) {
 try {
 let data = await parseJsonBody(req)
 res.json({ received: data })
 } catch (error) {
 res.status(400).json({ error: error.message })
 }
}
```

PARSING FORM DATA

URL-encoded form data:

```
function parseFormBody(req) {
 return new Promise((resolve, reject) => {
 let body = ""

 req.on("data", chunk => body += chunk)

 req.on("end", () => {
 let params = new URLSearchParams(body)
 resolve(Object.fromEntries(params))
 })

 req.on("error", reject)
 })
}
```

Request body: name=Alice&email=alice@example.com

Result: { name: "Alice", email: "alice@example.com" }

CONTENT TYPE DETECTION

```
function parseBody(req) {
 let contentType = req.headers["content-type"] ?? ""

 if (contentType.includes("application/json")) {
 return parseJsonBody(req)
 }

 if (contentType.includes("application/x-www-form-urlencoded")) {
 return parseFormBody(req)
 }

 return parseRawBody(req)
}

function parseRawBody(req) {
 return new Promise((resolve, reject) => {
 let chunks = []

 req.on("data", chunk => chunks.push(chunk))

 req.on("end", () => {
 resolve(Buffer.concat(chunks))
 })

 req.on("error", reject)
 })
}
```

QUERY PARAMETERS

```
function getQueryParams(req) {
 let url = new URL(req.url, `http://${req.headers.host}`)
 return Object.fromEntries(url.searchParams)
}

app.get("/search", (req, res) => {
 let query = getQueryParams(req)

 console.log(query.q)
 console.log(query.page)
 console.log(query.limit)

 res.json({ results: [], query })
})
```

Request: /search?q=hello&page=1&limit=10

URL PARAMETERS

Extract from route patterns:

```
app.get("/users/:userId/posts/:postId", (req, res) => {
 let { userId, postId } = req.params

 res.json({ userId, postId })
})
```

Request: /users/123/posts/456

Result: { userId: "123", postId: "456" }

HEADERS

Read request headers:

```
function handleRequest(req, res) {
 let contentType = req.headers["content-type"]
 let authorization = req.headers.authorization
 let userAgent = req.headers["user-agent"]
 let accept = req.headers.accept
 let host = req.headers.host

 console.log("Headers:", req.headers)
}
```

Headers are lowercase in Node.js.

COOKIES

Parse cookies from header:

```
function parseCookies(req) {
 let cookies = {}
 let header = req.headers.cookie

 if (header) {
 header.split(";").forEach(cookie = > {
 let [name, value] = cookie.trim().split("=")
 cookies[name] = decodeURIComponent(value)
 })
 }

 return cookies
}

app.get("/", (req, res) = > {
 let cookies = parseCookies(req)

 console.log(cookies.sessionId)
```

```
})
```

SETTING COOKIES

```
function setCookie(res, name, value, options = {}) {
 let cookie = `${name}=${encodeURIComponent(value)}`

 if (options.maxAge) {
 cookie += `; Max-Age=${options.maxAge}`
 }

 if (options.path) {
 cookie += `; Path=${options.path}`
 }

 if (options.httpOnly) {
 cookie += "; HttpOnly"
 }

 if (options.secure) {
 cookie += "; Secure"
 }

 res.setHeader("Set-Cookie", cookie)
}

app.post("/login", (req, res) => {
 setCookie(res, "sessionId", "abc123", {
 maxAge: 3600,
 httpOnly: true,
 path: "/"
 })

 res.json({ success: true })
})
```

REQUEST VALIDATION

```
function validateUser(data) {
 let errors = []

 if (!data.name || typeof data.name !== "string") {
 errors.push("Name is required")
 }

 if (!data.email || !data.email.includes("@")) {
 errors.push("Valid email is required")
 }

 if (data.age !== undefined && typeof data.age !== "number") {
 errors.push("Age must be a number")
 }

 return errors
}

app.post("/users", async (req, res) => {
 let body = await parseJsonBody(req)
 let errors = validateUser(body)

 if (errors.length > 0) {
 res.status(400).json({ errors })
 return
 }

 res.status(201).json({ id: Date.now(), ...body })
})
```

HANDLING LARGE REQUESTS

Limit body size:

```
function parseBodyWithLimit(req, maxSize = 1024 * 1024) {
 return new Promise((resolve, reject) => {
 let size = 0
 let chunks = []

 req.on("data", chunk => {
 size += chunk.length

 if (size > maxSize) {
 req.destroy()
 reject(new Error("Request too large"))
 return
 }

 chunks.push(chunk)
 })

 req.on("end", () => {
 resolve(Buffer.concat(chunks).toString())
 })

 req.on("error", reject)
 })
}
```

CORS HANDLING

```
function handleCors(req, res) {
 res.setHeader("Access-Control-Allow-Origin", "*")
 res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT,
DELETE, OPTIONS")
 res.setHeader("Access-Control-Allow-Headers", "Content-Type,
Authorization")
 res.setHeader("Access-Control-Max-Age", "86400")

 if (req.method === "OPTIONS") {
```

```
 res.writeHead(204)
 res.end()
 return true
 }

 return false
}

let server = http.createServer((req, res) => {
 if (handleCors(req, res)) {
 return
 }

 handleRequest(req, res)
})
```

ERROR RESPONSES

```
function sendError(res, status, message, details = null) {
 res.writeHead(status, { "Content-Type": "application/json" })

 res.end(JSON.stringify({
 error: {
 status,
 message,
 details
 }
 }))
}

app.get("/users/:id", (req, res) => {
 let user = findUser(req.params.id)

 if (!user) {
 sendError(res, 404, "User not found")
 return
 }
```

```
 res.json(user)
})
```

REQUEST TIMEOUT

```
let server = http.createServer((req, res) => {
 req.setTimeout(30000, () => {
 res.writeHead(408)
 res.end("Request Timeout")
 })

 handleRequest(req, res)
})
```

Or set on server:

```
server.timeout = 30000
```

PRACTICE

Parse different content types.

Handle query and URL parameters.

Work with headers and cookies.

Validate request data.

Handle CORS and errors.

WHAT YOU LEARNED

You learned to parse JSON and form bodies.
You learned query and URL parameters.
You learned to read and set headers.
You learned cookie handling.
You learned request validation.
You learned CORS handling.
You learned error response patterns.

The next chapter teaches you about serving files.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 18: Serving Files

SERVING STATIC FILES

Serve files from a directory:

```
const http = require("http")
const fs = require("fs")
const path = require("path")

let server = http.createServer((req, res) => {
 let filePath = path.join(__dirname, "public", req.url)

 fs.readFile(filePath, (error, content) => {
 if (error) {
 if (error.code === "ENOENT") {
 res.writeHead(404)
 res.end("File not found")
 } else {
 res.writeHead(500)
 res.end("Server error")
 }
 return
 }
```

```
 res.writeHead(200)
 res.end(content)
 })
})

server.listen(3000)
```

CONTENT TYPES

Set correct content type based on extension:

```
const mimeTypes = {
 ".html": "text/html",
 ".css": "text/css",
 ".js": "application/javascript",
 ".json": "application/json",
 ".png": "image/png",
 ".jpg": "image/jpeg",
 ".gif": "image/gif",
 ".svg": "image/svg+xml",
 ".ico": "image/x-icon",
 ".txt": "text/plain",
 ".pdf": "application/pdf",
 ".woff": "font/woff",
 ".woff2": "font/woff2"
}

function getMimeType(filePath) {
 let ext = path.extname(filePath).toLowerCase()
 return mimeTypes[ext] ?? "application/octet-stream"
}
```

COMPLETE STATIC FILE SERVER

```
const http = require("http")
const fs = require("fs").promises
const path = require("path")

async function serveFile(filePath, res) {
 try {
 let stats = await fs.stat(filePath)

 if (stats.isDirectory()) {
 filePath = path.join(filePath, "index.html")
 stats = await fs.stat(filePath)
 }

 let content = await fs.readFile(filePath)
 let mimeType = getMimeType(filePath)

 res.writeHead(200, {
 "Content-Type": mimeType,
 "Content-Length": stats.size
 })

 res.end(content)
 } catch (error) {
 if (error.code === "ENOENT") {
 res.writeHead(404, { "Content-Type": "text/html" })
 res.end("<h1>404 Not Found</h1>")
 } else {
 res.writeHead(500, { "Content-Type": "text/html" })
 res.end("<h1>500 Internal Server Error</h1>")
 }
 }
}

let server = http.createServer((req, res) => {
 let safePath = path.normalize(req.url).replace(/^(\.\.[\/\\])+/, "")
 let filePath = path.join(__dirname, "public", safePath)

 serveFile(filePath, res)
})
```

```
server.listen(3000, () => {
 console.log("Static server running on http://localhost:3000")
})
```

STREAMING LARGE FILES

Do not load large files into memory:

```
const fs = require("fs")

function serveFileStream(filePath, res) {
 let stat = fs.statSync(filePath)
 let mimeType = getMimeType(filePath)

 res.writeHead(200, {
 "Content-Type": mimeType,
 "Content-Length": stat.size
 })

 let stream = fs.createReadStream(filePath)

 stream.pipe(res)

 stream.on("error", (error) => {
 console.error("Stream error:", error)
 res.end()
 })
}
```

RANGE REQUESTS

Support partial content for video/audio:

```
function serveRangeRequest(filePath, req, res) {
 let stat = fs.statSync(filePath)
 let fileSize = stat.size
 let range = req.headers.range

 if (range) {
 let parts = range.replace(/bytes=/, "").split("-")
 let start = parseInt(parts[0], 10)
 let end = parts[1] ? parseInt(parts[1], 10) : fileSize - 1

 let chunkSize = end - start + 1

 res.writeHead(206, {
 "Content-Range": `bytes ${start}-${end}/${fileSize}`,
 "Accept-Ranges": "bytes",
 "Content-Length": chunkSize,
 "Content-Type": getMimeType(filePath)
 })

 let stream = fs.createReadStream(filePath, { start, end })

 stream.pipe(res)
 } else {
 res.writeHead(200, {
 "Content-Length": fileSize,
 "Content-Type": getMimeType(filePath)
 })

 fs.createReadStream(filePath).pipe(res)
 }
}
```

CACHING HEADERS

Add cache control:

```
function serveWithCache(filePath, res) {
```

```
 let mimeType = getMimeType(filePath)
 let maxAge = 86400

 res.writeHead(200, {
 "Content-Type": mimeType,
 "Cache-Control": `public, max-age=${maxAge}`,
 "ETag": generateETag(filePath)
 })
}

function generateETag(filePath) {
 let stat = fs.statSync(filePath)
 return `"${stat.size}-${stat.mtime.getTime()}"`
}
```

Check if modified:

```
function handleCaching(filePath, req, res) {
 let etag = generateETag(filePath)
 let clientEtag = req.headers["if-none-match"]

 if (clientEtag === etag) {
 res.writeHead(304)
 res.end()
 return true
 }

 return false
}
```

COMPRESSION

Compress text files:

```
const zlib = require("zlib")

function serveCompressed(filePath, req, res) {
```

```
 let acceptEncoding = req.headers["accept-encoding"] ?? ""
 let mimeType = getMimeType(filePath)
 let stream = fs.createReadStream(filePath)

 if (mimeType.startsWith("text/") || mimeType.includes("javascript")
|| mimeType.includes("json")) {
 if (acceptEncoding.includes("gzip")) {
 res.writeHead(200, {
 "Content-Type": mimeType,
 "Content-Encoding": "gzip"
 })

 stream.pipe(zlib.createGzip()).pipe(res)
 return
 }

 if (acceptEncoding.includes("deflate")) {
 res.writeHead(200, {
 "Content-Type": mimeType,
 "Content-Encoding": "deflate"
 })

 stream.pipe(zlib.createDeflate()).pipe(res)
 return
 }
 }

 res.writeHead(200, { "Content-Type": mimeType })
 stream.pipe(res)
}
```

FILE DOWNLOADS

Force download with Content-Disposition:

```
function serveDownload(filePath, downloadName, res) {
 let stat = fs.statSync(filePath)
```

```
 res.writeHead(200, {
 "Content-Type": "application/octet-stream",
 "Content-Disposition": `attachment;
filename="${downloadName}"`,
 "Content-Length": stat.size
 })

 fs.createReadStream(filePath).pipe(res)
}

app.get("/download/:filename", (req, res) => {
 let filePath = path.join(__dirname, "files", req.params.filename)

 if (fs.existsSync(filePath)) {
 serveDownload(filePath, req.params.filename, res)
 } else {
 res.status(404).end("File not found")
 }
})
```

SECURITY CONSIDERATIONS

Prevent directory traversal:

```
function sanitizePath(requestUrl, baseDir) {
 let requestPath = decodeURIComponent(requestUrl)

 requestPath = path.normalize(requestPath).replace(/^(\.\.[\/\
\])+/, "")

 let fullPath = path.join(baseDir, requestPath)

 if (!fullPath.startsWith(baseDir)) {
 return null
 }
```

```
 return fullPath
}

let filePath = sanitizePath(req.url, path.join(__dirname, "public"))

if (!filePath) {
 res.writeHead(403)
 res.end("Forbidden")
 return
}
```

SERVING INDEX FILES

```
function serveDirectory(dirPath, res) {
 let indexPath = path.join(dirPath, "index.html")

 if (fs.existsSync(indexPath)) {
 serveFile(indexPath, res)
 } else {
 res.writeHead(403)
 res.end("Directory listing not allowed")
 }
}
```

404 AND ERROR PAGES

```
async function serve404(res) {
 let errorPage = path.join(__dirname, "public", "404.html")

 try {
 let content = await fs.readFile(errorPage)
 res.writeHead(404, { "Content-Type": "text/html" })
 res.end(content)
 } catch {
```

```
 res.writeHead(404, { "Content-Type": "text/html" })
 res.end("<h1>404 Not Found</h1>")
 }
}
```

PRACTICE

Serve static files with correct MIME types.

Stream large files.

Implement caching.

Handle range requests.

Secure against path traversal.

WHAT YOU LEARNED

You learned to serve static files.
You learned MIME type handling.
You learned to stream large files.
You learned range requests for media.
You learned caching and compression.
You learned security considerations.

The next chapter teaches you about environment variables.

WHY ENVIRONMENT VARIABLES

Environment variables store configuration outside your code.

Benefits:
- Keep secrets out of code
- Different values for different environments
- Easy deployment configuration
- Twelve-factor app compliance

Never commit passwords, API keys, or secrets to git.

ACCESSING ENVIRONMENT VARIABLES

All environment variables are in process.env:

console.log(process.env.PATH)
console.log(process.env.HOME)
console.log(process.env.USER)
console.log(process.env.NODE_ENV)

They are always strings or undefined.

SETTING WHEN RUNNING

Set variables before the command:

PORT=3000 node app.js

API_KEY=secret DATABASE_URL=postgres://localhost/db node app.js

Multiple variables:

NODE_ENV=production PORT=8080 DEBUG=false node app.js

Access in code:

```
let port = process.env.PORT ?? "3000"
let apiKey = process.env.API_KEY
let nodeEnv = process.env.NODE_ENV ?? "development"
```

DOTENV FILES

Store variables in .env file:

```
NODE_ENV=development
PORT=3000
DATABASE_URL=postgres://localhost/mydb
API_KEY=your-secret-key
SECRET=another-secret
```

Install dotenv:

```
npm install dotenv
```

Load at start of app:

```
require("dotenv").config()

console.log(process.env.PORT)
console.log(process.env.API_KEY)
```

DOTENV WITH ES MODULES

```
import dotenv from "dotenv"

dotenv.config()
```

Or load early:

node --require dotenv/config app.js

DIFFERENT ENVIRONMENTS

Create environment-specific files:

.env Development defaults
.env.local Local overrides (gitignored)
.env.development Development settings
.env.production Production settings
.env.test Test settings

Load based on NODE_ENV:

```
const dotenv = require("dotenv")

let envFile = process.env.NODE_ENV === "production"
 ? ".env.production"
 : ".env.development"

dotenv.config({ path: envFile })
```

CONFIGURATION MODULE

Create a config module:

```
require("dotenv").config()

const config = {
 env: process.env.NODE_ENV ?? "development",
 port: parseInt(process.env.PORT ?? "3000", 10),
```

```
 database: {
 url: process.env.DATABASE_URL,
 pool: parseInt(process.env.DB_POOL_SIZE ?? "10", 10)
 },
 api: {
 key: process.env.API_KEY,
 secret: process.env.API_SECRET
 },
 jwt: {
 secret: process.env.JWT_SECRET,
 expiresIn: process.env.JWT_EXPIRES ?? "1h"
 },
 isDevelopment: process.env.NODE_ENV !== "production",
 isProduction: process.env.NODE_ENV === "production"
}

module.exports = config
```

Use throughout app:

```
const config = require("./config")

app.listen(config.port)

let db = connect(config.database.url)
```

REQUIRED VARIABLES

Check required variables at startup:

```
function requireEnv(name) {
 let value = process.env[name]

 if (!value) {
 console.error(`Missing required environment variable: ${name}`)
 process.exit(1)
 }
```

```
 return value
}

const config = {
 databaseUrl: requireEnv("DATABASE_URL"),
 apiKey: requireEnv("API_KEY"),
 port: process.env.PORT ?? "3000"
}
```

VALIDATION

Validate and transform values:

```
function getEnvInt(name, defaultValue) {
 let value = process.env[name]

 if (!value) return defaultValue

 let parsed = parseInt(value, 10)

 if (isNaN(parsed)) {
 throw new Error(`${name} must be a number`)
 }

 return parsed
}

function getEnvBool(name, defaultValue = false) {
 let value = process.env[name]

 if (!value) return defaultValue

 return value.toLowerCase() === "true" || value === "1"
}

let port = getEnvInt("PORT", 3000)
```

```
let debug = getEnvBool("DEBUG", false)
```

.GITIGNORE

Never commit secrets:

```
.env
.env.local
.env.*.local
*.env
```

Do commit example file:

.env.example

Example file shows what variables are needed without actual values:

```
NODE_ENV=development
PORT=3000
DATABASE_URL=your-database-url
API_KEY=your-api-key
```

SECRETS IN PRODUCTION

Production options:

Platform environment variables:
Heroku, Vercel, AWS, etc. have settings for env vars.

Secret managers:
AWS Secrets Manager, HashiCorp Vault, etc.

Docker:

Pass through docker run -e or docker-compose.yml.

Never hardcode production secrets.

DOCKER ENVIRONMENT

In Dockerfile:

```
ENV NODE_ENV=production
ENV PORT=3000
```

In docker-compose.yml:

```
services:
 app:
 environment:
 - NODE_ENV=production
 - PORT=3000
 - DATABASE_URL=${DATABASE_URL}
```

Pass from host:

```
docker run -e API_KEY=secret myapp
```

CROSS-PLATFORM SCRIPTS

Windows handles env vars differently.

Use cross-env package:

```
npm install --save-dev cross-env
```

In package.json:

```
{
 "scripts": {
 "start": "cross-env NODE_ENV=production node app.js",
 "dev": "cross-env NODE_ENV=development node app.js"
 }
}
```

Works on Windows, macOS, and Linux.

DEBUGGING ENVIRONMENT

Print all variables (be careful with secrets):

```
console.log(process.env)
```

Print specific ones:

```
console.log("Environment:", process.env.NODE_ENV)
console.log("Port:", process.env.PORT)
console.log("DB connected:", !!process.env.DATABASE_URL)
```

Check what is set without revealing values:

```
Object.keys(process.env)
 .filter(key => key.startsWith("MY_APP_"))
 .forEach(key => {
 console.log(`${key}: ${process.env[key] ? "[SET]" : "[NOT SET]"}`)
 })
```

PRACTICE

Use dotenv for local development.

Create configuration module.

Validate required variables.

Keep secrets out of code.

Use environment-specific files.

WHAT YOU LEARNED

You learned to access environment variables.
You learned dotenv for local development.
You learned configuration module pattern.
You learned to validate and require variables.
You learned gitignore for secrets.
You learned cross-platform considerations.

The next chapter teaches you about debugging Node.js.
BOOK 13: NODE.JS FUNDAMENTALS
Chapter 20: Debugging Node.js

CONSOLE METHODS

console.log for general output:

```
console.log("Value:", value)
console.log("User:", { name: "Alice", age: 25 })
```

console.error for errors:

```
console.error("Error:", error.message)
```

console.warn for warnings:

console.warn("Deprecation warning")

console.table for tabular data:

```
console.table([
 { name: "Alice", age: 25 },
 { name: "Bob", age: 30 }
])
```

console.time and timeEnd for timing:

```
console.time("operation")
doSomething()
console.timeEnd("operation")
```

Output: operation: 123.456ms

console.trace for stack trace:

```
console.trace("Where am I?")
```

INSPECT FOR DEEP OBJECTS

```
const util = require("util")

let deepObject = { a: { b: { c: { d: { e: "deep" } } } } }

console.log(util.inspect(deepObject, {
 depth: null,
 colors: true
}))
```

Or use console.dir:

```
console.dir(deepObject, { depth: null })
```

DEBUG MODULE

Install debug for controlled logging:

npm install debug

Use in code:

```
const debug = require("debug")("app:server")
const dbDebug = require("debug")("app:db")

debug("Server starting")
dbDebug("Database connected")
```

Enable with environment variable:

DEBUG=app:* node app.js

DEBUG=app:server node app.js

DEBUG=app:server,app:db node app.js

NODE INSPECTOR

Built-in debugger with Chrome DevTools:

node --inspect app.js

Output: Debugger listening on ws://127.0.0.1:9229/...

Open Chrome and go to:

chrome://inspect

Click "inspect" to open DevTools.

Break on first line:

node --inspect-brk app.js

VS CODE DEBUGGING

Create .vscode/launch.json:

```
{
 "version": "0.2.0",
 "configurations": [
 {
 "type": "node",
 "request": "launch",
 "name": "Launch Program",
 "program": "${workspaceFolder}/app.js",
 "skipFiles": ["<node_internals>/**"]
 },
 {
 "type": "node",
 "request": "attach",
 "name": "Attach to Process",
 "port": 9229
 }
 ]
}
```

Press F5 to start debugging.

DEBUGGER STATEMENT

Add breakpoints in code:

```
function processData(data) {
 debugger

 let result = transform(data)

 debugger

 return result
}
```

When running with --inspect, execution pauses at debugger.

ERROR STACK TRACES

Get full stack traces:

```
Error.stackTraceLimit = Infinity
```

Or set limit:

```
Error.stackTraceLimit = 50
```

Get clean stack traces:

```
function cleanStack(error) {
 return error.stack
 .split("\n")
 .filter(line => !line.includes("node_modules"))
 .join("\n")
}
```

ASYNC STACK TRACES

Node.js 12+ has async stack traces by default.

For earlier versions or more detail:

node --async-stack-traces app.js

Shows full async call history.

MEMORY DEBUGGING

Check memory usage:

```
function logMemory() {
 let used = process.memoryUsage()

 console.log({
 rss: `${Math.round(used.rss / 1024 / 1024)} MB`,
 heapTotal: `${Math.round(used.heapTotal / 1024 / 1024)} MB`,
 heapUsed: `${Math.round(used.heapUsed / 1024 / 1024)} MB`,
 external: `${Math.round(used.external / 1024 / 1024)} MB`
 })
}

setInterval(logMemory, 5000)
```

Create heap snapshot:

node --inspect app.js

In Chrome DevTools Memory tab, take heap snapshot.

PROFILING

CPU profiling:

node --prof app.js

Processes the log:

node --prof-process isolate-*.log > profile.txt

Or use --inspect and Chrome DevTools Performance tab.

CATCHING UNCAUGHT ERRORS

```
process.on("uncaughtException", (error) => {
 console.error("Uncaught Exception:")
 console.error(error.stack)
 process.exit(1)
})

process.on("unhandledRejection", (reason, promise) => {
 console.error("Unhandled Rejection at:", promise)
 console.error("Reason:", reason)
})
```

LOGGING BEST PRACTICES

Structured logging:

```
function log(level, message, meta = {}) {
 console.log(JSON.stringify({
 timestamp: new Date().toISOString(),
 level,
 message,
 ...meta
 }))
}
```

```
log("info", "Server started", { port: 3000 })
log("error", "Database connection failed", { error: "timeout" })
```

Output:

```
{"timestamp":"2024-01-15T10:30:00.000Z","level":"info","message":"Server started","port":3000}
```

Use logging libraries for production:

```
npm install winston
```

```
const winston = require("winston")

let logger = winston.createLogger({
 level: "info",
 format: winston.format.json(),
 transports: [
 new winston.transports.File({ filename: "error.log", level: "error" }),
 new winston.transports.File({ filename: "combined.log" })
 ]
})

if (process.env.NODE_ENV !== "production") {
 logger.add(new winston.transports.Console({
 format: winston.format.simple()
 }))
}
```

DEBUGGING HTTP REQUESTS

Log all requests:

```
app.use((req, res, next) => {
 let start = Date.now()

 res.on("finish", () => {
 console.log({
```

```
 method: req.method,
 url: req.url,
 status: res.statusCode,
 duration: Date.now() - start
 })
 })

 next()
})
```

COMMON DEBUGGING SCENARIOS

Finding where error occurs:

```
try {
 problematicCode()
} catch (error) {
 console.error("Error in problematicCode:")
 console.error(error.stack)
}
```

Tracking variable changes:

```
let _value

Object.defineProperty(globalThis, "trackedValue", {
 get() {
 console.log("Getting trackedValue:", _value)
 console.trace()
 return _value
 },
 set(newValue) {
 console.log("Setting trackedValue:", newValue)
 console.trace()
 _value = newValue
 }
})
```

Finding slow operations:

```
async function timed(name, fn) {
 console.time(name)
 let result = await fn()
 console.timeEnd(name)
 return result
}

await timed("database query", () => db.query(sql))
```

PRACTICE

Use console methods effectively.

Set up VS Code debugging.

Use debug module for conditional logging.

Handle uncaught errors.

Profile memory and CPU.

WHAT YOU LEARNED

You learned console methods for output.
You learned Node Inspector with Chrome.
You learned VS Code debugging setup.
You learned async stack traces.
You learned memory debugging.
You learned logging best practices.

The next chapter brings everything together with practice programs.

BOOK 13: NODE.JS FUNDAMENTALS

Chapter 21: Practice Node Programs

FILE MANAGER CLI

Command-line file management tool:

```
const fs = require("fs").promises
const path = require("path")
const readline = require("readline")

class FileManager {
 constructor() {
 this.currentDir = process.cwd()
 }

 async list(dir = ".") {
 let fullPath = path.resolve(this.currentDir, dir)
 let items = await fs.readdir(fullPath, { withFileTypes: true })

 let files = []
 let dirs = []

 items.forEach(item => {
 if (item.isDirectory()) {
 dirs.push(item.name + "/")
 } else {
 files.push(item.name)
 }
 })

 return [...dirs.sort(), ...files.sort()]
 }

 async cd(dir) {
```

```
let newPath = path.resolve(this.currentDir, dir)
let stats = await fs.stat(newPath)

if (!stats.isDirectory()) {
throw new Error("Not a directory")
}

this.currentDir = newPath
return this.currentDir
}

async read(file) {
let fullPath = path.resolve(this.currentDir, file)
return fs.readFile(fullPath, "utf8")
}

async write(file, content) {
let fullPath = path.resolve(this.currentDir, file)
await fs.writeFile(fullPath, content)
return fullPath
}

async mkdir(name) {
let fullPath = path.resolve(this.currentDir, name)
await fs.mkdir(fullPath, { recursive: true })
return fullPath
}

async remove(name) {
let fullPath = path.resolve(this.currentDir, name)
await fs.rm(fullPath, { recursive: true })
return fullPath
}

async copy(source, dest) {
let srcPath = path.resolve(this.currentDir, source)
let destPath = path.resolve(this.currentDir, dest)
await fs.copyFile(srcPath, destPath)
return destPath
}
```

```
 async move(source, dest) {
 let srcPath = path.resolve(this.currentDir, source)
 let destPath = path.resolve(this.currentDir, dest)
 await fs.rename(srcPath, destPath)
 return destPath
 }

 async search(pattern, dir = ".") {
 let results = []
 let regex = new RegExp(pattern, "i")

 async function searchDir(currentPath) {
 let items = await fs.readdir(currentPath, { withFileTypes: true })

 for (let item of items) {
 let fullPath = path.join(currentPath, item.name)

 if (regex.test(item.name)) {
 results.push(fullPath)
 }

 if (item.isDirectory()) {
 await searchDir(fullPath)
 }
 }
 }

 await searchDir(path.resolve(this.currentDir, dir))
 return results
 }
}

async function main() {
 let fm = new FileManager()

 let rl = readline.createInterface({
 input: process.stdin,
 output: process.stdout
 })
```

```
function prompt() {
rl.question(`${fm.currentDir}> `, async (input) => {
let [cmd, ...args] = input.trim().split(" ")

try {
switch (cmd) {
case "ls":
let items = await fm.list(args[0])
items.forEach(item => console.log(item))
break
case "cd":
await fm.cd(args[0])
break
case "cat":
console.log(await fm.read(args[0]))
break
case "mkdir":
await fm.mkdir(args[0])
break
case "rm":
await fm.remove(args[0])
break
case "cp":
await fm.copy(args[0], args[1])
break
case "mv":
await fm.move(args[0], args[1])
break
case "search":
let results = await fm.search(args[0])
results.forEach(r => console.log(r))
break
case "exit":
rl.close()
return
default:
console.log("Unknown command")
}
} catch (error) {
```

```
 console.error("Error:", error.message)
 }

 prompt()
 })
 }

 prompt()
}

main()
```

SIMPLE HTTP API SERVER

```
const http = require("http")
const fs = require("fs").promises

class ApiServer {
 constructor() {
 this.routes = new Map()
 this.data = { users: [], posts: [] }
 }

 route(method, path, handler) {
 this.routes.set(`${method}:${path}`, handler)
 }

 async loadData() {
 try {
 let content = await fs.readFile("data.json", "utf8")
 this.data = JSON.parse(content)
 } catch {
 this.data = { users: [], posts: [] }
 }
 }

 async saveData() {
```

```
await fs.writeFile("data.json", JSON.stringify(this.data, null, 2))
}

parseBody(req) {
return new Promise((resolve) => {
let body = ""
req.on("data", chunk => body += chunk)
req.on("end", () => {
try {
resolve(body ? JSON.parse(body) : null)
} catch {
resolve(null)
}
})
})
}

matchRoute(method, url) {
let pathname = new URL(url, "http://localhost").pathname

for (let [key, handler] of this.routes) {
let [routeMethod, routePath] = key.split(":")

if (routeMethod !== method) continue

let params = {}
let routeParts = routePath.split("/")
let urlParts = pathname.split("/")

if (routeParts.length !== urlParts.length) continue

let match = true

for (let i = 0; i < routeParts.length; i++) {
if (routeParts[i].startsWith(":")) {
params[routeParts[i].slice(1)] = urlParts[i]
} else if (routeParts[i] !== urlParts[i]) {
match = false
break
}
```

```
}

if (match) return { handler, params }
}

return null
}

async handleRequest(req, res) {
res.json = (data, status = 200) => {
res.writeHead(status, { "Content-Type": "application/json" })
res.end(JSON.stringify(data))
}

let match = this.matchRoute(req.method, req.url)

if (!match) {
res.json({ error: "Not found" }, 404)
return
}

req.params = match.params
req.body = await this.parseBody(req)

try {
await match.handler(req, res, this.data)
await this.saveData()
} catch (error) {
res.json({ error: error.message }, 500)
}
}

listen(port) {
let server = http.createServer((req, res) => {
this.handleRequest(req, res)
})

server.listen(port, async () => {
await this.loadData()
console.log(`API server running on http://localhost:${port}`)
```

```
 })
 }
}

let api = new ApiServer()

api.route("GET", "/users", (req, res, data) => {
 res.json(data.users)
})

api.route("GET", "/users/:id", (req, res, data) => {
 let user = data.users.find(u => u.id === req.params.id)
 user ? res.json(user) : res.json({ error: "Not found" }, 404)
})

api.route("POST", "/users", (req, res, data) => {
 let user = { id: Date.now().toString(), ...req.body }
 data.users.push(user)
 res.json(user, 201)
})

api.route("DELETE", "/users/:id", (req, res, data) => {
 let index = data.users.findIndex(u => u.id === req.params.id)
 if (index >= 0) {
 data.users.splice(index, 1)
 res.json({ success: true })
 } else {
 res.json({ error: "Not found" }, 404)
 }
})

api.listen(3000)
```

LOG FILE ANALYZER

```
const fs = require("fs")
const readline = require("readline")
```

```
const path = require("path")

class LogAnalyzer {
 constructor() {
 this.stats = {
 totalLines: 0,
 levels: {},
 errors: [],
 byHour: {},
 patterns: {}
 }
 }

 async analyze(logFile) {
 let stream = fs.createReadStream(logFile)

 let rl = readline.createInterface({
 input: stream,
 crlfDelay: Infinity
 })

 for await (let line of rl) {
 this.processLine(line)
 }

 return this.getReport()
 }

 processLine(line) {
 this.stats.totalLines++

 let match = line.match(/\[(\d{4}-\d{2}-\d{2}T\d{2}:\d{2}:
\d{2})\]\s*\[(\w+)\]\s*(.*)/)

 if (!match) return

 let [, timestamp, level, message] = match

 this.stats.levels[level] = (this.stats.levels[level] ?? 0) + 1
```

```
 let hour = timestamp.slice(11, 13)
 this.stats.byHour[hour] = (this.stats.byHour[hour] ?? 0) + 1

 if (level === "ERROR") {
 this.stats.errors.push({ timestamp, message })
 }

 let errorType = message.match(/^(\w+Error):/)?.[1]

 if (errorType) {
 this.stats.patterns[errorType] = (this.stats.patterns[errorType] ??
0) + 1
 }
 }

 getReport() {
 return {
 summary: {
 totalLines: this.stats.totalLines,
 levels: this.stats.levels,
 errorCount: this.stats.errors.length
 },
 errorsByHour: this.stats.byHour,
 recentErrors: this.stats.errors.slice(-10),
 errorPatterns: this.stats.patterns
 }
 }
}

async function main() {
 let logFile = process.argv[2]

 if (!logFile) {
 console.log("Usage: node log-analyzer.js <logfile>")
 process.exit(1)
 }

 let analyzer = new LogAnalyzer()
 let report = await analyzer.analyze(logFile)
```

```
 console.log("\n=== Log Analysis Report ===\n")
 console.log("Total Lines:", report.summary.totalLines)
 console.log("\nLog Levels:")
 Object.entries(report.summary.levels).forEach(([level, count]) =>
{
 console.log(`  ${level}: ${count}`)
 })

 console.log("\nErrors by Hour:")
 Object.entries(report.errorsByHour)
 .sort(([a], [b]) => a.localeCompare(b))
 .forEach(([hour, count]) => {
 console.log(`  ${hour}:00 - ${count} entries`)
 })

 if (report.recentErrors.length > 0) {
 console.log("\nRecent Errors:")
 report.recentErrors.forEach(err => {
 console.log(`  [${err.timestamp}] ${err.message}`)
 })
 }

 if (Object.keys(report.errorPatterns).length > 0) {
 console.log("\nError Patterns:")
 Object.entries(report.errorPatterns)
 .sort(([, a], [, b]) => b - a)
 .forEach(([pattern, count]) => {
 console.log(`  ${pattern}: ${count}`)
 })
 }
}

main()
```

STATIC FILE SERVER WITH LIVE RELOAD

```
const http = require("http")
```

```
const fs = require("fs")
const path = require("path")

const mimeTypes = {
 ".html": "text/html",
 ".css": "text/css",
 ".js": "application/javascript",
 ".json": "application/json",
 ".png": "image/png",
 ".jpg": "image/jpeg",
 ".svg": "image/svg+xml"
}

class LiveServer {
 constructor(root, port = 3000) {
 this.root = path.resolve(root)
 this.port = port
 this.clients = []
 }

 injectLiveReload(html) {
 let script = `
 <script>
 let es = new EventSource("/_live");
 es.onmessage = () => location.reload();
 </script>
 `
 return html.replace("</body>", script + "</body>")
 }

 serveFile(filePath, res) {
 let ext = path.extname(filePath)
 let mime = mimeTypes[ext] ?? "application/octet-stream"

 fs.readFile(filePath, (err, data) => {
 if (err) {
 res.writeHead(404)
 res.end("Not found")
 return
 }
```

```
if (ext === ".html") {
data = this.injectLiveReload(data.toString())
}

res.writeHead(200, { "Content-Type": mime })
res.end(data)
})
}

handleSSE(res) {
res.writeHead(200, {
"Content-Type": "text/event-stream",
"Cache-Control": "no-cache",
"Connection": "keep-alive"
})

this.clients.push(res)

res.on("close", () => {
this.clients = this.clients.filter(c => c !== res)
})
}

notifyClients() {
this.clients.forEach(client => {
client.write("data: reload\n\n")
})
}

watchFiles() {
fs.watch(this.root, { recursive: true }, (event, filename) => {
if (filename && !filename.startsWith(".")) {
console.log(`File changed: ${filename}`)
this.notifyClients()
}
})
}

start() {
```

```
 let server = http.createServer((req, res) => {
 if (req.url === "/_live") {
 this.handleSSE(res)
 return
 }

 let urlPath = req.url === "/" ? "/index.html" : req.url
 let filePath = path.join(this.root, urlPath)

 if (!filePath.startsWith(this.root)) {
 res.writeHead(403)
 res.end("Forbidden")
 return
 }

 this.serveFile(filePath, res)
 })

 server.listen(this.port, () => {
 console.log(`Live server: http://localhost:${this.port}`)
 console.log(`Serving: ${this.root}`)
 })

 this.watchFiles()
 }
}

let root = process.argv[2] ?? "."
let port = parseInt(process.argv[3] ?? "3000", 10)

new LiveServer(root, port).start()
```

WHAT YOU BUILT

You built four complete Node.js programs:

File manager CLI with commands

REST API server with persistence
Log file analyzer
Live reload static server

Each program demonstrated:

File system operations
HTTP server creation
Streams and events
Command-line interfaces
Real-world patterns

WHAT COMES NEXT

You now understand Node.js fundamentals:

- What Node.js is (Book 13, Chapter 1-2)
- Running JavaScript (Book 13, Chapter 3)
- Modules and NPM (Book 13, Chapters 4-6)
- Process and environment (Book 13, Chapters 7, 19)
- File system (Book 13, Chapters 8-11)
- Paths (Book 13, Chapter 12)
- Streams and events (Book 13, Chapters 13-14)
- HTTP servers (Book 13, Chapters 15-18)
- Debugging (Book 13, Chapter 20)

Book 14 teaches you Express.js and backend development. You will:

- Build APIs with Express.js
- Use middleware patterns
- Handle authentication
- Connect to databases
- Deploy applications

You finished Book 13. You run JavaScript on servers. Now you build real backends.